CS6
Practical System Skills
Fall 2019 edition
Leonhard Spiegelberg
lspiegel@cs.brown.edu

16.99 Recap

Last lectures: Basic + more advanced python

- **Basic:**
  Comments, Numbers, Strings, Lists, Tuples, Dictionaries, Variables, Functions, Lambdas, Slicing, String formatting, Control Flow(if/for/while), builtin functions
- **Advanced:**
  Comprehensions(list/set/dict), Decorators, Generators, Higher order functions(map/filter), Basic I/O, Modules

17 Flask

CS6 Practical System Skills

Fall 2019

Leonhard Spiegelberg *lspiegel@cs.brown.edu*

17.01 What is Flask?

⇒ lightweight python framework to quickly build web applications

—> there are many other popular python frameworks like Django, Bottle, Falcon, web2py, ...

⇒ many extensions available for Flask for forms, databases, authentication...

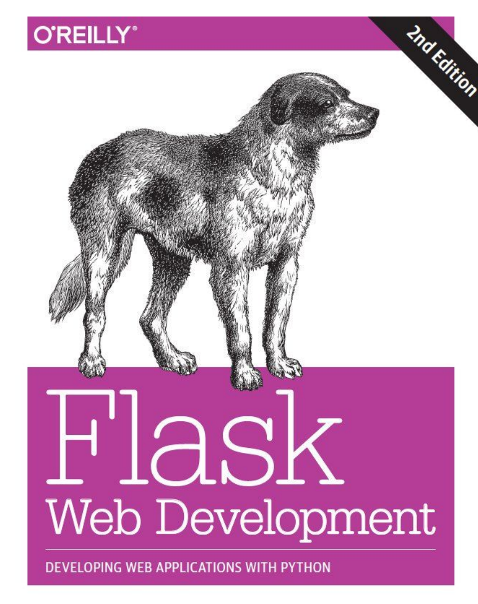
⇒ `pip3 install flask`

⇒ all examples from today available @ github.com/browncs6/FlaskExamples

17.02 Flask resources

Book:

Flask Web Development by *Miguel Grinberg*. ISBN: 9781491991732



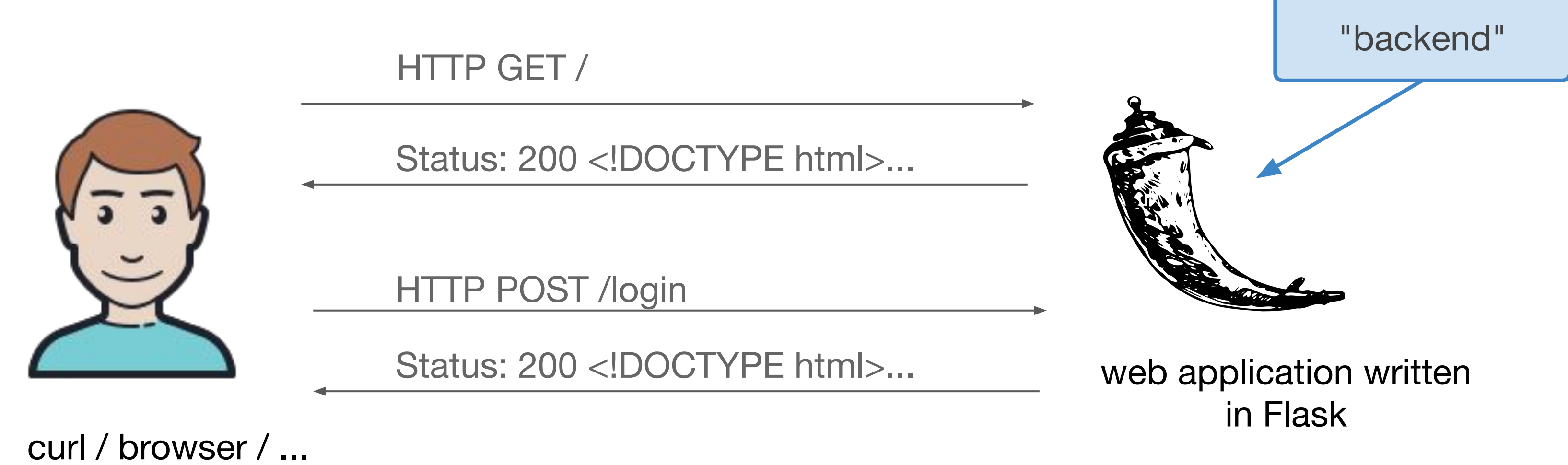
Websites:

http://flask.palletsprojects.com/en/1.1.x/

http://exploreflask.com/en/latest/

https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world

17.03 The big picture



⇒ a user requests (one or more) resource(s) via (one or more) URIs

—> web application written in Flask responds with content

17.04 Why use Flask?

⇒ allows to create dynamic websites, i.e. return dynamic content and process requests!

⇒ static vs. dynamic websites == fixed vs. dynamic content

—> a static website delivers always the same content to any user

⇒ What are examples for static and dynamic websites?

17.04 Static vs. Dynamic websites

Static	Dynamic
API Documentation	search engine
Blog (without comments or Disqus)	online tax program
News page	Banner
...	...

⇒ most websites are actually dynamic web applications. To create static websites, use a static website generator like Jekyll!

17.05 A hello world application in flask

hw.py

```python
from flask import Flask

app = Flask(__name__)

# define routes here
@app.route('/')
def index():
    return '<h1>Hello world</h1>'


if __name__ == '__main__':
    app.run()
```

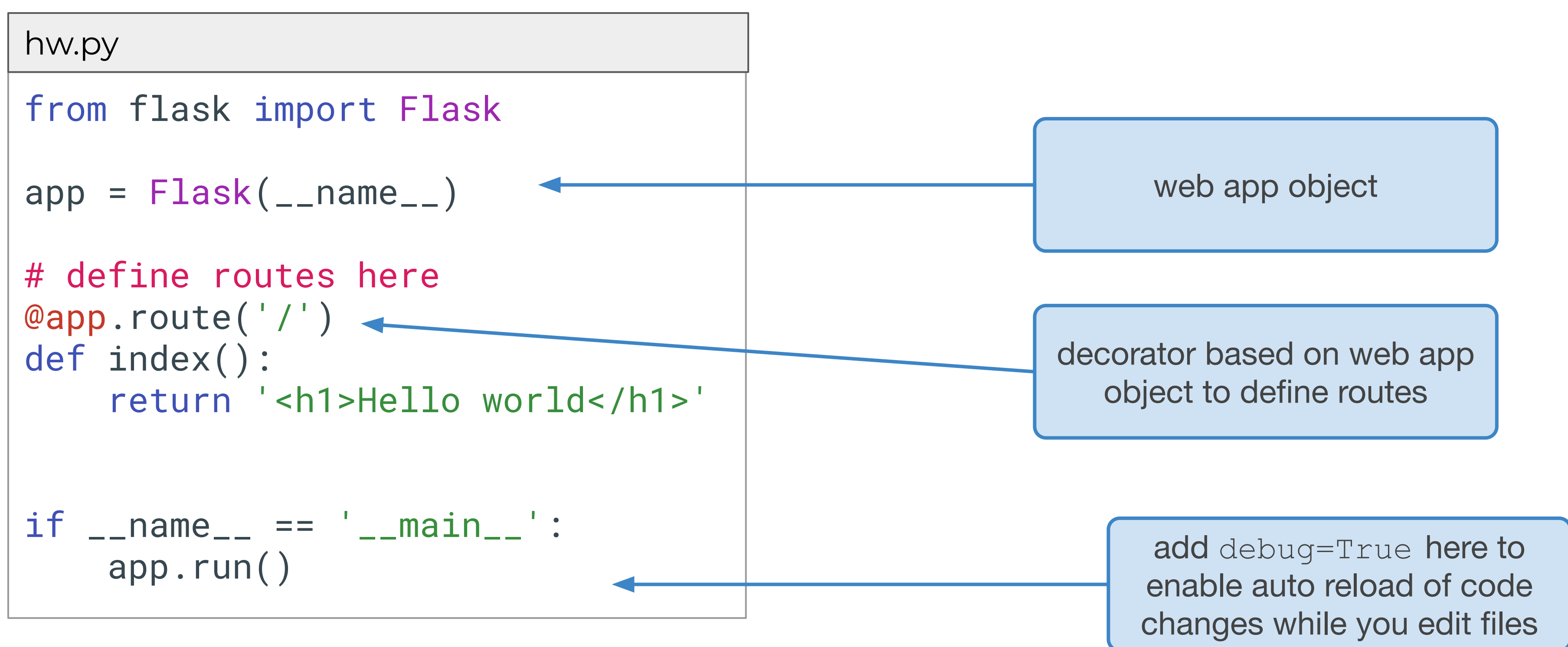
start via

```
python3 hw.py
```

or via

```
export FLASK_APP=hw.py
&& flask run
```

17.05 A detailed look



17.06 Defining routes in Flask

⇒ Basic idea: Return content for a route (i.e. the path + query segment of an URI, e.g. `/blog/12/03/09`)

⇒ Flask can assign parts of the urls to python variables!

```
@app.route('/blog/<int:year>/<int:month>/<title>')
def blog(title, month, year):
    return '...'
```

Syntax is `<varname>`

Note: the order doesn't matter in the python function. Optionally, a flask filter can be applied to make sure the URI part is of certain type

17.06 Defining routes in Flask

```
@app.route('/blog/<int:year>/<int:month>/<title>')
def blog(title, month, year):
    return 'Blog entry from {}/{}'.format(month, year)
```

types for Flask routes	
`string`	accepts any text without a slash (default)
`int`	accepts integers
`float`	accepts numerical values containing decimal points
`path`	similar to a string but accepts slashes

17.07 Responses and error codes

⇒ each HTTP response comes with a status code. You can explicitly define them in your Flask application:

```
@app.route('/404')
def make404():
    return 'This page yields a 404 error', 404
```

no content will be displayed, because 404 error!

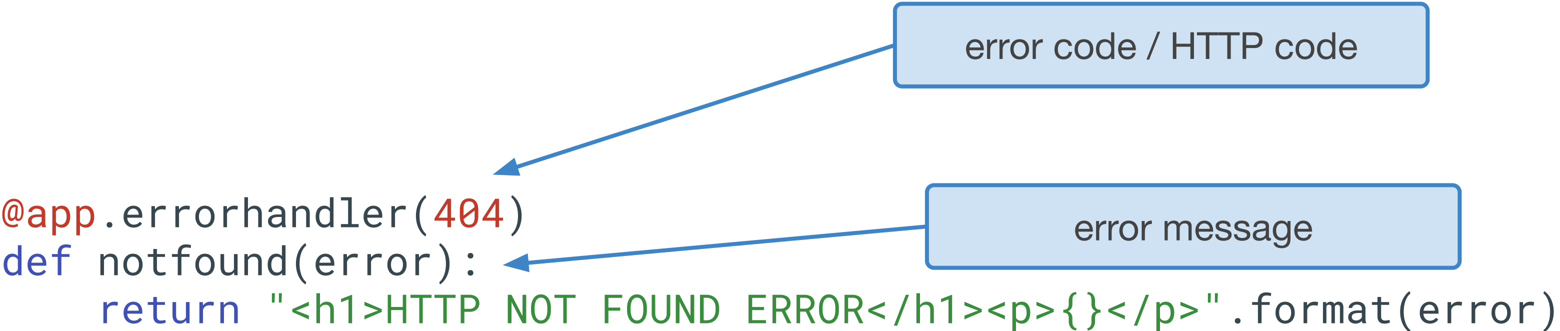
⇒ for complete list of HTTP status codes and their meaning confer https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

⇒ per default Flask returns 200 (HTTP OK)

⇒ if you defined a route with variables/types whose constraints are violated, Flask will responds with 404

17.07 HTTP error pages

⇒ in order to display a custom error page for a certain HTTP code, you can define a route in Flask via `app.errorhandler()`



⇒ instead of explicitly returning, you may also use Flask's builtin function `abort(code, description=)` to leave a route with an error code

17.08 Request and Response objects

⇒ when defining a route via flask, within the function a request object is available holding details of the HTTP request

⇒ to create a response, a response object needs to be returned. Flask has several helper functions to create response objects, when returning a string per default it is treated as HTML response.

⇒ There are multiple types of requests (e.g. GET/POST), via `methods=[...]` keyword argument a route can be restricted to work only for specific requests.

17.08 Request object

```
from flask import request

@app.route('/get',methods=['GET'])
def get_route():
    response = '<p>{} request {} issued<p><p>' \
               'Headers<br>{}</p>' \
               '<p>Query args:<br>{}'.format(request.method,
                                              request.full_path,
                                              request.headers,
                                              request.args)
    return response, 200
```

⇒ test via `curl` `http://localhost:5000/get?a=10&b=30`
or by entering a URL to this route to the browser

17.08 Response object

```
@app.route('/post', methods=['POST'])
def post_route():

    body = '<table>'
    for k, v in request.form.items():
        body = '<tr><td>{}</td><td>{}</td></tr>'.format(k, v)
    body += '</table>'

    response = make_response(body)
    response.headers['X-Parachutes'] = 'parachutes are cool'
    return response
```

⇒ a post request can be issued via curl, e.g.

```
curl -sD - --form 'name=tux' \
           --form 'profession=penguin' http://localhost:5000/post
```

17.09 URIs and responses

⇒ Note that a URI can return any content, i.e. you can also generate images on-the-fly, csv files, videos, …

⇒ specify MIME-type (MIME=) when creating response

List of MIME types: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types

Example:

Return a csv file via route `/csv`

syntax alternative to make_response

```
@app.route('/csv')
def csv():
    return app.response_class(response='a,b,c\n1,2,3\n4,5,6',
                              mimetype='text/csv')
```

Templating + static files

17.10 Templates

⇒ so far: custom written responses

—> cumbersome, better to use templates and fill in stubs!

⇒ Flask comes with a powerful template engine.

—> specify template path via
`Flask(__name__, template_folder=...)`
—> default template folder is `templates/`
—> in addition to templates, Flask can serve static files.
(Option: `static_folder`, default folder: `static/`)

⇒ **Core idea:** When creating a response, load template and fill in placeholders using data from backend!

17.10 Jinja2 templates

⇒ Flask uses Jinja2 as template engine
https://jinja.palletsprojects.com/en/2.10.x/

⇒ Jinja2 has two types of stubs:

- `{{ ... }}` print result of expression to template
- `{% ... %}` execute statements, e.g. loops/if/...

⇒ use in flask via

```
render_template('home.html', title='Hello world')
```

stored in `templates/` folder

pass variable named title to template with content `'Hello world'`

17.10 Jinja2 template language

⇒ `{{ expression }}` replaces `{{ … }}` with the value of the expression
expression can be something like `2 * var`, `data.name` or a function registered to Jinja2

⇒ `{% for … %} … {% endfor %}`
allows to create complex HTML structure quickly

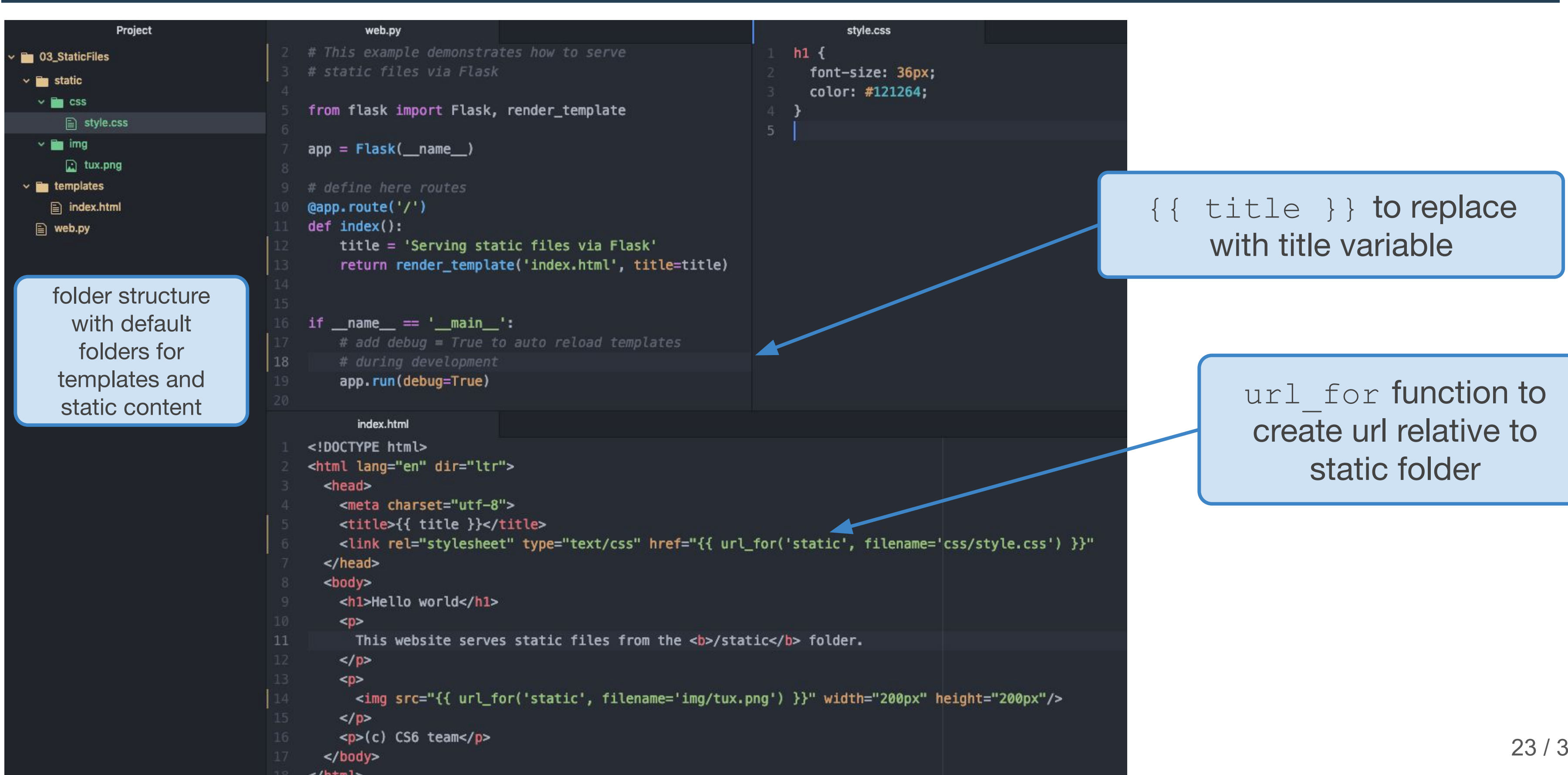
⇒ `{% if … %} … {% endif %}`
allows to create HTML code depending on condition

⇒ documentation under https://jinja.palletsprojects.com/en/2.10.x/templates/

Example:

```
<ul>
{% for user in users %}
  <li><a href="{{ user.url }}">{{ user.username }}</a></li>
{% endfor %}
</ul>
```

17.10 Jinja2 templates in Flask - example

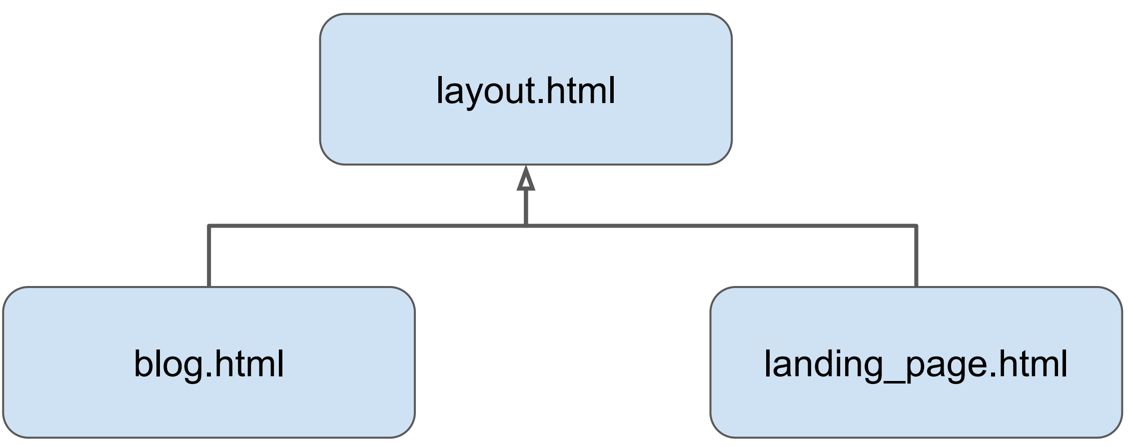


17.11 More on templates

⇒ Jinja2 provides many helpful mechanisms like filters, macros, ...

⇒ Especially useful is template inheritance.

—> use `{% extends layout.html %}` and `{% block name %} ... {% endblock %}` for inheritance



17.12 Template inheritance example

layout.html

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title>Template inheritance
example</title>
  </head>
  <body>
    {% block body %}
    <h1>Parent</h1>
    <!-- empty template !-->
    {% endblock %}

    <p>q.e.d.</p>
  </body>
</html>
```

child.html

```
{% extends 'layout.html' %}
{% block body %}
<h1>Child example</h1>
Block of parent replaced
with this content here.
{% endblock %}
```

`child.html` inherits from `layout.html`. Jinja2 replaces the body block of the parent with the content of child's one.

HTML forms

17.13 Making websites interactive via forms

⇒ User input can be captured using forms

—> good resource on this topic:

Jon Duckett's HTML/CSS book Chapters 7 and 14

⇒ To define a form create one or more input fields `<input>` enclosed in `<form>...</form>`tags.

⇒ When the submit button of a form is clicked a GET or POST request will be issued to the URI defined under action.

—> input fields with a `name` attribute will be encoded.

17.13 General structure of input elements

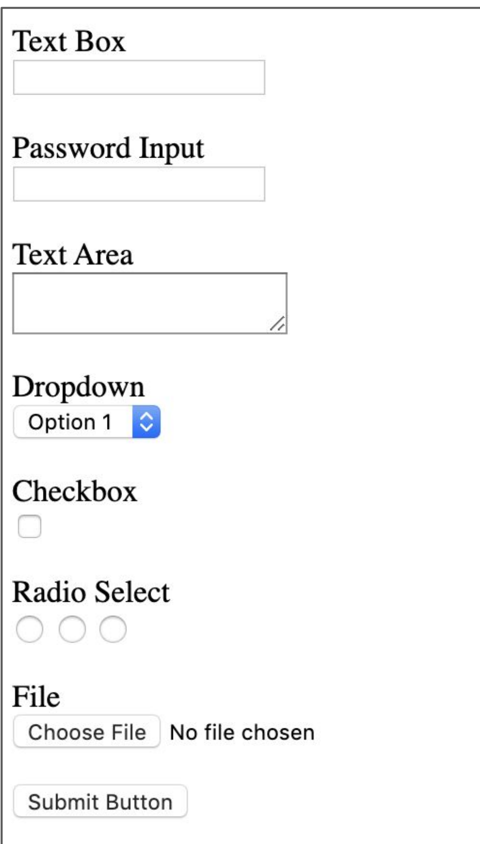
```
<input type="..." name="..." value="...">
```

type of the input field, can be e.g. text, submit, checkbox, … A list is available under https://www.w3schools.com/html/html_form_input_types.asp

When a form is submitted, data is encoded as name=value pairs.

17.13 forms example

```
<form action="/dest" method="GET">
<label for="textbox">Text Box</label><br>
<input type="text" name="textbox"><br><br>
<label for="password">Password Input</label><br>
<input type="password" name="password"><br><br>
<label for="textbox">Text Area</label><br>
<textarea name="textarea"></textarea><br><br>
<label for="dropdown">Dropdown</label><br>
<select id="dropdown">
<option value="1">Option 1</option>
<option value="2">Option 2</option>
<option value="3">Option 3</option>
</select><br><br>
<label for="checkbox">Checkbox</label><br>
<input type="checkbox" name="checkbox"><br><br>
<label for="radio">Radio Select</label><br>
<input type="radio" name="radio">
<input type="radio" name="radio">
<input type="radio" name="radio"><br><br>
<label for="file">File</label><br>
<input type="file" name="file"><br><br>
<input type="submit" value="Submit Button">
</form>
```



17.13 Data encoding

⇒ Depending on the method (POST or GET) specified, the data of the form is encoded in one of the following ways:

- When `GET` is used, the data becomes part of the URI `/calc?first_operand=12&second_operand=3`
- When `POST` is used, the data is encoded in the body of the request message

```
<form action="/calc" method="get">
    <input type="text" name="first_operand" value="0">
    <select name="operator">
    <option value="+">+</option>
    <option value="-">-</option>
    <option value="*">*</option>
    <option value="/">/</option>
    </select>
    <input type="text" name="second_operand" value="0"><br><br>
    <input type="submit" id="submit" value="calculate" />
  </form>
```

Example form

17.14 Accessing form data in Flask

⇒ Flask allows to easily access form data when a URI is invoked.

—> you can also simulate forms via `curl` and `--form/-F` option!

```
@app.route('/calc', methods=['POST'])
def calc():
    res = 'undefined'
    op1 = int(request.form['first_operand'])
    op2 = int(request.form['second_operand'])
    op = request.form['operator']

    if op == '+':
        res = op1 + op2
    elif op == '-':
        res = op1 - op2
    elif op == '*':
        res = op1 * op2
    elif op == '/':
        res = op1 / op2
    else:
        abort(404)
    return render_template(...)
```

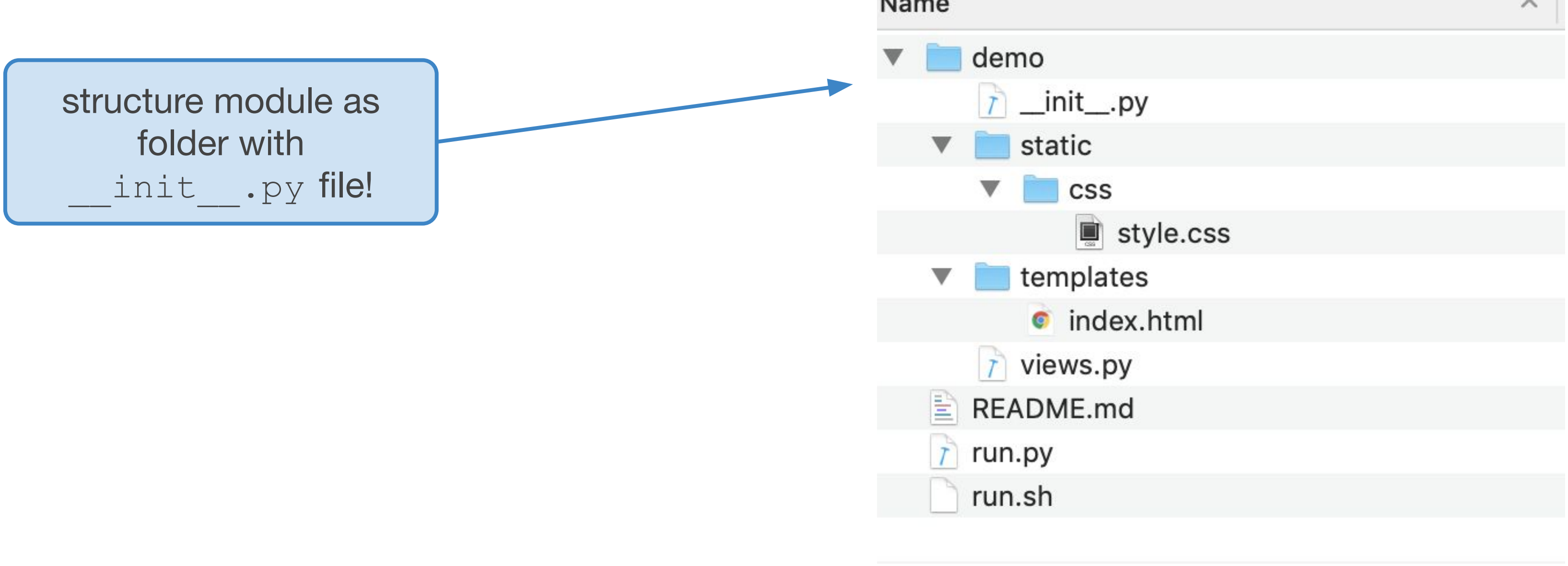
extract form elements from form dictionary

More complex flask applications

17.07 Organizing the module

⇒ So far: single .py file which held all logic.

⇒ A more complicated project might need multiple files! How to structure them?



17.08 Next steps

⇒ There are many extensions available for flask, e.g.

- `flask-login`	provides decorators to secure routes
- `flask-mail`	sending emails
- `flask-cache`	cache routes
- `flask-pymongo`	connect to MongoDB database
- `flask-mysql`	connect to MySQL database
- `flask-wtf`	wtforms integration (validators & Co)
- ...	

⇒ When developing your final project, some of these might be helpful!

End of lecture.

Next class: Thu, 4pm-5:20pm @ CIT 477