MAKE SCHOOL

# Templating

WEB 1.1

# Agenda

- Learning Outcomes
- Review: Forms
- Why Templating?
- Variables in Templates
- **BREAK**
- If Statements
- Loops
- Template Inheritance
- Wrap-Up

# Learning Outcomes

By the end of today, you should be able to...

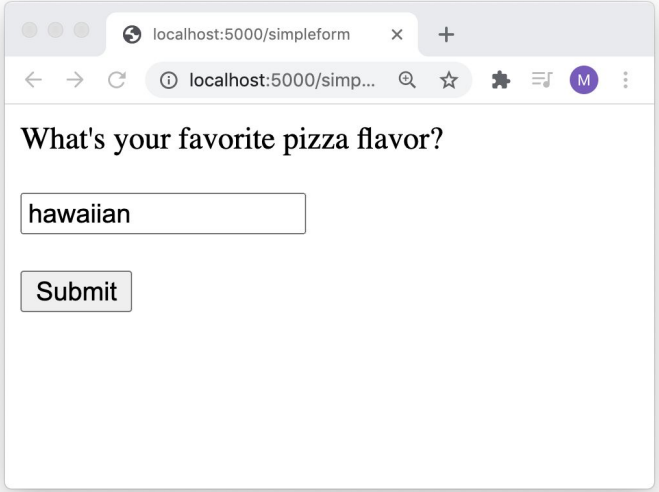
1. **Explain** why templating is important for making our code more elegant & readable.
2. **Use** Jinja2 templates to display variables, conditionals, & list items.
3. **Use** template inheritance to re-use code on multiple pages.

# Review: Forms

# Form Essential Parts

Any form on the Web has 3 essential parts: The **<form> tags**, at least one **input tag**, and a **submit button**.

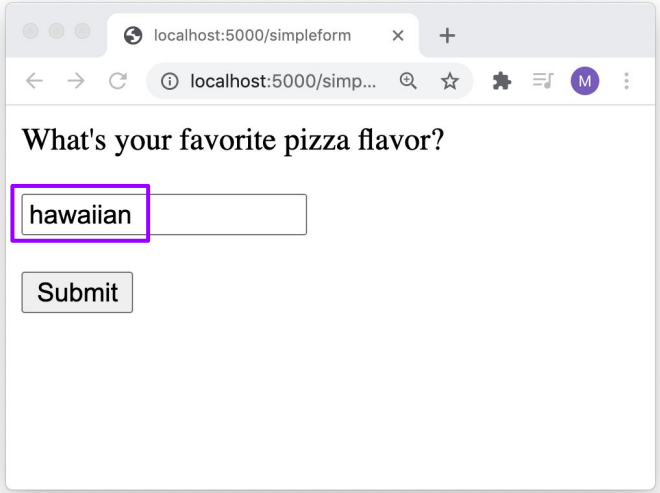
```
<form>
   What's your favorite pizza flavor?
   <input type="text" name="pizza_flavor">
   <input type="submit" value="Submit">
</form>
```



# Form Essential Parts

The **name** field is the **key** of the **key-value pair** that is sent to the server. *If the name field is missing, no data will be sent!*

```
<form>
   What's your favorite pizza flavor?
   <input type="text" name="pizza_flavor">
   <input type="submit" value="Submit">
</form>
```




client → pizza_flavor=hawaiian → server

# Form Essential Parts

There are 2 form attributes that are usually included: the **action** and the **method**.

Where are these results going?

```
<form action="/results" method="GET">
   What's your favorite pizza flavor?
   <input type="text" name="pizza_flavor">
   <input type="submit" value="Submit">
</form>
```

How are they being sent there?

The **action** says *which URL* to send the results to.

The **method** says *how to send* the results. It can only be **GET** or **POST** (GET is the default). We'll talk about the differences between these later.

# GET vs. POST

When to use the **GET** method vs. the **POST** method?

- Use **GET** to request data or information.
  - Example: sending a search query in a search form
- Use **POST** to send data to update a resource, or otherwise change the state of the system.
  - Example: sending username/password in a login form
  - Example: creating a new database object

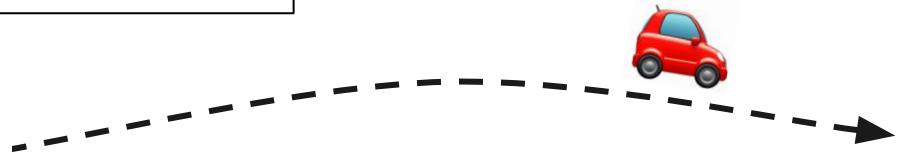
GET vs POST diagram

# Form Results

**`request.args`** is like a **suitcase** that holds all of the **key-value pairs** that were typed in by the user. It only works for **GET** routes.

```
email=me@me.com
phone=3334445555
crust=thick
size=10
toppings=pineapple
toppings=onions
accepted_terms=yes
```

request.args



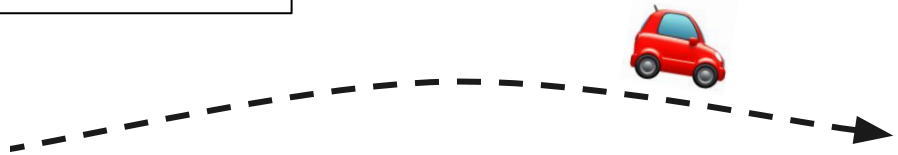
<form method="GET">

# Form Results

`request.form` works exactly the same way, but only works for **POST** routes.

```
email=me@me.com
phone=3334445555
crust=thick
size=10
toppings=pineapple
toppings=onions
accepted_terms=yes
```

request.form



<form method="POST">

# Activity (15 minutes)

With a partner, complete the TODOs in the Forms Repl.it to get the results of the form, perform a calculation, and display to the user.

Make sure to practice good Pair Programming! The **Driver** should share their screen, and the **Navigator** should instruct them on what to type.

# Why Templating?

# Templating

Let's take a look at some of the code we've written so far.

The route for `/pizza/submit` combines both HTML and Python code.

This has worked well so far, because it keeps everything in one place, but once our projects start to get larger and larger, it's a really bad practice.

What are some **downsides** to having our code in one file?

```
@app.route('/pizza/submit', methods=['GET', 'POST'])
def submit_pizza():
    users_email = request.args.get('email')
    users_phone = request.args.get('phone')
    crust_type = request.args.get('crust')
    pizza_size = request.args.get('size')
    list_of_toppings = request.args.getlist('toppings')
    accepted_terms = request.args.get('terms_conditions')

    if accepted_terms != 'accepted':
        return 'Please accept the terms and conditions and try again!'

    return f"""
    Your order summary: <br>
    Email: {users_email} <br>
    Phone number: {users_phone} <br><br>

    You ordered a {crust_type} crust pizza of size {pizza_size}-inch
    with the following toppings: {', '.join(list_of_toppings)}
    """
```

# Templating

Here are some **downsides** to having **Python** and **HTML** code in the same file:

- There's no syntax highlighting! So it's really hard to see what's going on.
- The code editor can't make autocomplete suggestions, like telling us when an end-bracket is missing.
- It just looks messy!

So... what can we do instead??

```
@app.route('/pizza/submit', methods=['GET', 'POST'])
def submit_pizza():
    users_email = request.args.get('email')
    users_phone = request.args.get('phone')
    crust_type = request.args.get('crust')
    pizza_size = request.args.get('size')
    list_of_toppings = request.args.getlist('toppings')
    accepted_terms = request.args.get('terms_conditions')

    if accepted_terms != 'accepted':
        return 'Please accept the terms and conditions and try again!'

    return f"""
    Your order summary: <br>
    Email: {users_email} <br>
    Phone number: {users_phone} <br><br>

    You ordered a {crust_type} crust pizza of size {pizza_size}-inch
    with the following toppings: {', '.join(list_of_toppings)}
    """
```

# Templating

MAKE SCHOOL

We want to place all of our HTML code for each page into a **separate HTML file**.

This will enforce the convention of **only having one programming language** in a single file, which will make our code cleaner and more elegant!

We'll be using the **render_template** function (built into Flask) to accomplish this.

# Ah, much better!

MAKE SCHOOL

## app.py

```python
@app.route('/pizza/submit', methods=['GET', 'POST'])
def submit_pizza():
    accepted_terms = request.args.get('terms_conditions')
    if accepted_terms != 'accepted':
        return render_template('toc_not_accepted.html')

    list_of_toppings = request.args.getlist('toppings')
    context = {
        users_email: request.args.get('email'),
        users_phone: request.args.get('phone'),
        crust_type: request.args.get('crust'),
        pizza_size: request.args.get('size'),
        toppings: ', '.join(list_of_toppings)
    }

    return render_template('submission_page.html', **context)
```

## templates/toc_not_accepted.html

```html
<p>
    Please accept the terms and conditions and try
again!
<p>
```

## templates/submission_page.html

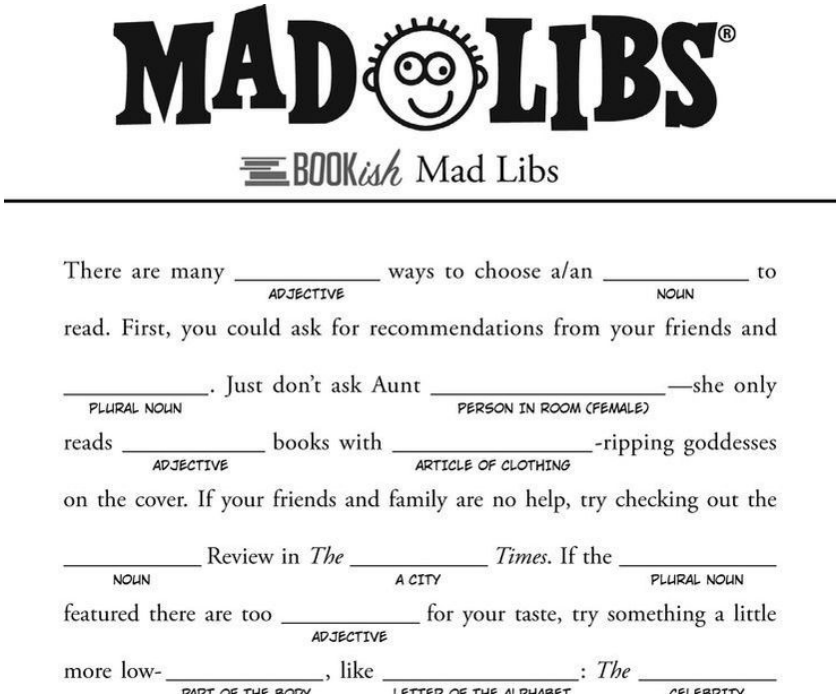
```jinja
Your order summary: <br>
Email: {{ users_email }} <br>
Phone number: {{ users_phone }} <br><br>

You ordered a {{ crust_type }} crust pizza
of size {{ pizza_size }}-inch
with the following toppings: {{ toppings }}
```

Python code

HTML code (Jinja2)

# What are Templates?

A template is kind of like a game of MadLibs - we pass variables in from the *route* to be used in the *template*.



MAD LIBS®

BOOKish Mad Libs

There are many _____ ways to choose a/an _____ to
ADJECTIVE NOUN
read. First, you could ask for recommendations from your friends and
_____. Just don't ask Aunt _____—she only
PLURAL NOUN PERSON IN ROOM (FEMALE)
reads _____ books with _____-ripping goddesses
ADJECTIVE ARTICLE OF CLOTHING
on the cover. If your friends and family are no help, try checking out the
_____ Review in *The* _____ *Times*. If the _____
NOUN A CITY PLURAL NOUN
featured there are too _____ for your taste, try something a little
ADJECTIVE
more low-_____, like _____: *The* _____
PART OF THE BODY LETTER OF THE ALPHABET CELEBRITY

# Let’s break it down...

# Jinja2 Templating

The templating language we'll be using is called **Jinja2**. When **render_template** is called, the Jinja2 template tags get **transformed into regular HTML** before being sent to the client.

The **context** variable here is like a suitcase that **packages up variables** to be used in the template.

Jinja2 uses {{ }} syntax to denote using a variable.

app.py

```
def submit_pizza():
    ...
    context = {
        users_email: request.args.get('email'),
        users_phone: request.args.get('phone'),
        crust_type: request.args.get('crust'),
        pizza_size: request.args.get('size'),
        toppings: ', '.join(list_of_toppings)
    }

    return render_template('submission_page.html', **context)
```

templates/submission_page.html

```
Your order summary: <br>
Email: {{ users_email }} <br>
Phone number: {{ users_phone }} <br><br>

You ordered a {{ crust_type }} crust pizza
of size {{ pizza_size }}-inch
with the following toppings: {{ toppings }}
```

# Jinja2 Templating

Every **key-value pair** in the context gets transformed into a **variable** to be used in the template.

*(Yes, variables are just key-value pairs!)*

app.py

```
def submit_pizza():
    ...
    context = {
        users_email: request.args.get('email'),
        users_phone: request.args.get('phone'),
        crust_type: request.args.get('crust'),
        pizza_size: request.args.get('size'),
        toppings: ', '.join(list_of_toppings)
    }

    return render_template('submission_page.html', **context)
```

templates/submission_page.html

```
Your order summary: <br>
Email: {{ users_email }} <br>
Phone number: {{ users_phone }} <br><br>

You ordered a {{ crust_type }} crust pizza
of size {{ pizza_size }}-inch
with the following toppings: {{ toppings }}
```

# Jinja2 Templating

Let’s try one more simple example together.

Let’s refactor 1 route in the Jinja Refactor Repl.lt.

# Jinja2 Templating Activity (10 minutes)

Refactor the rest of the routes in the Jinja Refactor Repl.It.

In breakout groups of 2, practice **Pair Programming** as you work through the routes:

- The **Driver** shares their screen & types in the code.
- The **Navigator** tells the driver what to type.
- Switch roles after each route.

# Break - 10 min

# Jinja2 If Statements

# If Statements

MAKE SCHOOL

Sometimes, we want to show something different in a template depending on the value of a variable.

Let's say we have an **Animal Facts** web page and we want to show a different fact for each animal.

We could have an **if/elif/else** statement in the Python code to determine which animal fact to show... **or** we could do it in the template!

# If Statements

Here's what the code might look like.

## app.py

```
@app.route('/fact/<animal>')
def animal_fact(animal):
    """Shows one fact about the given animal."""

    return render_template(
        'fact.html',
        animal=animal)
```

## templates/fact.html

```
{% if animal == 'aardvark' %}
   Aardvarks can eat up to 50,000 insects each night! They
swallow their food whole, without chewing it.
{% elif animal == 'penguin' %}
   A penguin's black and white plumage serves as camouflage while
swimming.
{% elif animal == 'zebra' %}
   A zebra's stripes serve to dazzle and confuse predators and
biting insects.
{% else %}
   I don't have any facts about that animal. Please try again!
{% endif %}
```

*If the user goes to /fact/koala, what will they see?*

# If Statements

An if statement in Jinja2 looks like this:

```
{% if boolean expression %}

   If the first statement is true, this HTML will be displayed!

{% elif other boolean expression %}

   If the first statement is true, this HTML will be displayed!

{% else %}

   If neither is true, this HTML will be displayed!

{% endif %}
```

Just like in Python, the *elif* and *else* clauses are optional.

# If Statements

With a partner (Pair Programming style), complete the [If Statements Repl.it]() by filling in the missing code in `templates/coin_flip.html`.

Some Flask Repl.it tips:

- If you see “Hmm.... We Couldn’t Reach Your Repl”, click “Stop” followed by “Run”.
- You can also click the icon to open in a new tab.
- Collect more tips to help out your classmates!

# Jinja2 Loops

# Loops

Sometimes we want to show an HTML element, or several elements, *multiple times* in one web page.

Instead of writing out the same code many times, we can use a *for loop* to accomplish this!

# Loops

Here's an example of a Jinja2 for loop in action! *What will we see rendered on the page?*

app.py

```
@app.route('/compliments')
def get_compliments():
    """Gives the user some compliments."""
    compliments = [
        'brave',
        'witty',
        'tenacious'
    ]

    return render_template(
        'compliments.html',
        compliments=compliments)
```

templates/compliments.html

```
Hello there, user! You are so...

<ul>
{% for compliment in compliments %}
    <li> {{ compliment }} </li>
{% endfor %}
</ul>
```

# Let's break it down!

Recall that the only variables available in the template are the ones added into the context - that is, `compliments_list`.

BUT, the for loop creates a new variable, `compliment`!

app.py

```
def get_compliments():
    ...

    return render_template(
        'compliments.html',
        compliments_list=compliments)
```

templates/compliments.html

```
Hello there, user! You are so...

<ul>
{% for compliment in compliments_list %}
    <li> {{ compliment }} </li>
{% endfor %}
</ul>
```

# Loops

A loop in Jinja looks like:

```
{% for new_variable_name in list %}

   This HTML will be repeatedly displayed for each item in the

list. Each time the loop runs, {{ new_variable_name }} will

refer to a different item in the list.

{% endfor %}
```

# Loops

With a partner (Pair Programming style), complete the Loops Repl.it by filling in the missing code in `templates/shopping_list.html`.

Some Flask Repl.it tips:

- If you see “Hmm.... We Couldn’t Reach Your Repl”, click “Stop” followed by “Run”.
- You can also click the icon to open in a new tab.

# Template Inheritance

# Templating Exploration Activity (10 minutes)

- Click on this web page and this one.
- List at least **3 things that are the same** and **3 things that are different** between the **layouts** of the two web pages. (Example: Both pages include a search bar.)
- Type your answer in the chat!

# Templating

MAKE SCHOOL

**Same**

- Nav bar
- Footer
- Styles

**Different**

- Content
- Etc...

# Why Templating?

In the programming world, there is a concept called

**D**on't

**R**epeat

**Y**ourself

which means, don't write the same code twice.

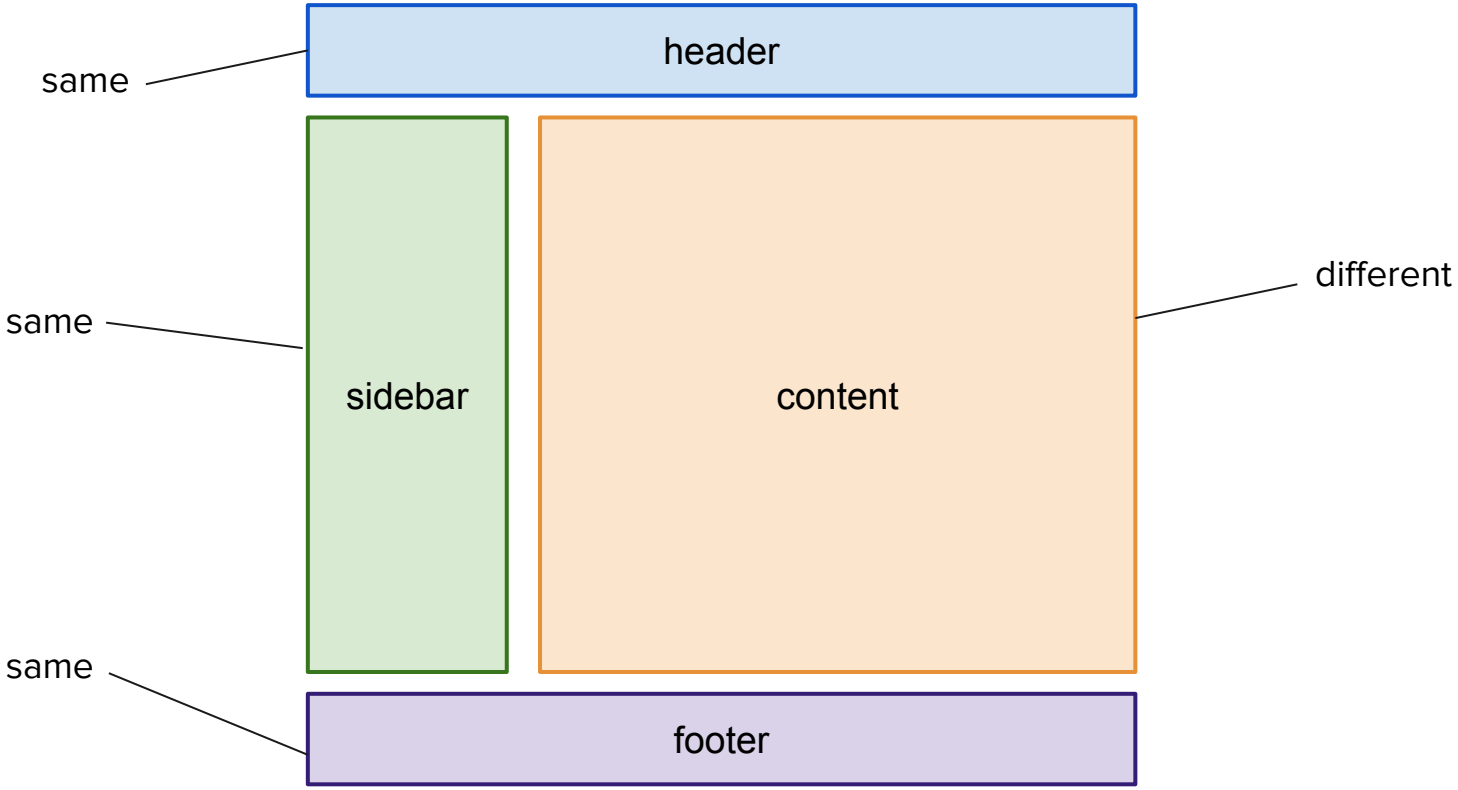
How does that concept relate to the web pages we just explored?

# Templating

It is very common to have parts of a web page that are re-used for every page on the website: header, sidebar, footer, etc.

We can use **template inheritance** to easily reuse parts of a web page in multiple places.

# Templating



# Templating

We can use the `{% block %}` and `{% extends … %}` tags to inherit from a base template.

## templates/base.html

```
<!DOCTYPE html>
<html>
   <head>
       <title>
           {% block title %}{% endblock %}
       </title>
   </head>
   <body>
       <header>
           ...
       </header>

       <main>
           {% block content %} PLACEHOLDER TEXT {% endblock %}
       </main>

       <footer>
           ...
       </footer>
   </body>
</html>
```

## templates/index.html

```
{% extends 'base.html' %}

{% block title %}
   My Pizza Delivery App
{% endblock %}

{% block content %}
   Welcome! Here is the place where you order pizza.
{% endblock %}
```

# Templating

Fill in the blanks to complete the `index.html` template!

## templates/pizza_base.html

```
<!DOCTYPE html>
<html>
    ...
    <body>

        <main>
            Please fill out the following form to order your
pizza.
            {% block pizza_form %}{% endblock %}
        </main>

    </body>
</html>
```

## templates/index.html

```
{% extends                    %}

{% block            %}
    <form action="/pizza_submit">
        Select your pizza crust:
        ...
        <input type="submit" value="Submit">
    </form>
{% endblock %}
```

# Templating

With a partner (Pair Programming style), complete the Template Inheritance Repl.it by filling in the missing code in `templates/home.html`.

# Lab Time: Homework 2

# Resources

- Templates - Flask Tutorial
- (Beginner-friendly) Video Tutorial on Templates