Flask Cheatsheet

A quick reference guide to the Flask micro web framework for Python, covering essential concepts, code snippets, and best practices.

Core Concepts

Basic Application Structure

```python
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'

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

This is the basic structure of a Flask application. `Flask(__name__)` creates the application instance. The `@app.route` decorator binds a URL route to a function.

The `app.run(debug=True)` starts the development server. Setting `debug=True` enables the debugger and reloader.

Routing

`@app.route('/path')`	Binds the function to the `/path` URL. You can define multiple routes for a single function.
`@app.route('/path/<variable>')`	Adds a variable part to the URL. The variable is passed as an argument to the function.
`@app.route('/path/<int:variable>')`	Specifies the type of the variable as an integer. Other options include `float` and `string` (default).
`methods=['GET', 'POST']`	Specifies the HTTP methods allowed for the route. Defaults to `GET`.

Request Object

```python
from flask import request

@app.route('/login', methods=['POST'])
def login():
    username = request.form['username']
    password = request.form['password']
    # ...
```

The `request` object provides access to incoming request data, such as form data (`request.form`), query parameters (`request.args`), and request headers (`request.headers`).

`request.method` - The HTTP method used for the request (e.g., 'GET', 'POST').

`request.url` - The full URL of the request.

Templates and Rendering

Rendering Templates

```python
from flask import render_template

@app.route('/hello/')
@app.route('/hello/<name>')
def hello(name=None):
    return render_template('hello.html',
name=name)
```

The `render_template` function renders a Jinja2 template. The first argument is the template filename, and subsequent arguments are variables passed to the template.

Templates are located in the `templates` directory by default.

Jinja2 Basics

`{{ variable }}`	Outputs the value of a variable.
`{% ... %}`	Executes a statement (e.g., loop, conditional).
`{# ... #}`	Comment.
`{{ url_for('function_name') }}`	Generates a URL for a function based on its route. Useful for avoiding hardcoded URLs.

Template Inheritance

`{% extends 'base.html' %}` - Extends a base template.
`{% block block_name %}` ... `{% endblock %}` - Defines a block that can be overridden in child templates.

Base template (`base.html`):

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

Child template (`index.html`):

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

{% block title %}Home{% endblock %}

{% block content %}
    <h1>Welcome!</h1>
{% endblock %}
```

Working with Databases

Flask-SQLAlchemy

Flask-SQLAlchemy simplifies using SQLAlchemy with Flask.

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] =
'sqlite:///test.db'
db = SQLAlchemy(app)

class User(db.Model):
    id = db.Column(db.Integer,
primary_key=True)
    username = db.Column(db.String(80),
unique=True, nullable=False)
    email = db.Column(db.String(120),
unique=True, nullable=False)

    def __repr__(self):
        return '<User %r>' % self.username
```

Configuration: `app.config['SQLALCHEMY_DATABASE_URI']` sets the database connection string. Use `sqlite:///test.db` for SQLite.

Defining Models

`db.Column(db.Integer, primary_key=True)`	Defines an integer primary key column.
`db.Column(db.String(80), unique=True)`	Defines a string column with a maximum length of 80 characters that must be unique.
`db.relationship`	Defines a relationship between two models.

Database Operations

```
# Create the database tables
with app.app_context():
    db.create_all()

# Create a new user
new_user = User(username='john_doe',
email='john@example.com')
db.session.add(new_user)
db.session.commit()

# Query users
users = User.query.all()
user =
User.query.filter_by(username='john_doe').firs
t()
```

`db.create_all()` creates the database tables.

`db.session.add()` adds a new object to the session. `db.session.commit()` commits the changes to the database.

Forms and Validation

Flask-WTF

Flask-WTF integrates WTForms with Flask for handling forms.

```
from flask_wtf import FlaskForm
from wtforms import StringField,
PasswordField, SubmitField
from wtforms.validators import DataRequired,
Email, EqualTo

class RegistrationForm(FlaskForm):
    username = StringField('Username',
validators=[DataRequired()])
    email = StringField('Email', validators=
[DataRequired(), Email()])
    password = PasswordField('Password',
validators=[DataRequired()])
    confirm_password = PasswordField('Confirm
Password', validators=[DataRequired(),
EqualTo('password')])
    submit = SubmitField('Register')
```

Define forms as classes inheriting from `FlaskForm`. Use `StringField`, `PasswordField`, etc., for form fields. Add validators like `DataRequired` and `Email`.

Using Forms in Views

```
from flask import render_template, redirect,
url_for

@app.route('/register', methods=['GET',
'POST'])
def register():
    form = RegistrationForm()
    if form.validate_on_submit():
        # Process the form data
        return redirect(url_for('success'))
    return render_template('register.html',
form=form)
```

`form.validate_on_submit()` validates the form data. If validation passes, the function returns `True`.

Displaying Forms in Templates

```
<form method="POST">
    {{ form.csrf_token }}
    <div class="form-group">
        {{ form.username.label }} {{
form.username(class="form-control") }}
        {% if form.username.errors %}
            <ul class="errors">
            {% for error in
form.username.errors %}
                <li>{{ error }}</li>
            {% endfor %}
            </ul>
        {% endif %}
    </div>
    <button type="submit">{{ form.submit.label
}}</button>
</form>
```

Use `{{ form.field_name.label }}` to display the field label and `{{ form.field_name(class="form-control") }}` to display the field input. Iterate over `form.field_name.errors` to display validation errors.