**[ORMs, APIs](https://cs50.harvard.edu/web/2018/notes/4/" \l "orms-apis)**

[**Object-Oriented Programming**](https://cs50.harvard.edu/web/2018/notes/4/#object-oriented-programming)

* Python, along with many other programming langauges, use Object-Oriented Programming (OOP). An ‘object’ is a discrete item. OOP allows for the creation of classes, which are the generic forms of objects. For example, a ‘flight’ class is defines all the components which describe a flight, as well as actions that a flight should be able to take, such as adding a passenger. Similarly, a passenger class would represent the generic idea of passenger, defined by a name and associated with a flight, perhaps.
* Here’s a simple example of a Python class.
* **class** **Flight**:
* **def** **\_\_init\_\_**(self, origin, destination, duration):
* self.origin **=** origin
* self.destination **=** destination
* self.duration **=** duration
  + \_\_init\_\_ is a ‘method’, which is a function performed on individual objects. \_\_init\_\_ in particular is a special, built-in method that describes what should happen when a flight object is created.
  + Generally, methods take self as their first argument. self refers to the object being worked with. The other three arguments are simply the information that should be stored about a particular flight. That information is stored as ‘properties’ inside the object, using dot notation.
* Here’s how the Flight class might be used:
* *# Create flight.*
* f **=** Flight(origin**=**"New York", destination**=**"Paris", duration**=**540)
* *# Change the value of a property.*
* f.duration **+=** 10
* *# Print details about flight.*
* **print**(f.origin)
* **print**(f.destination)
* **print**(f.duration)
  + Note that only flight information is passed in to Flight(); the self argument to the \_\_init\_\_ method is automatically specified.
  + f is a variable of type Flight, just like a variable might be of type str or int.
* Additional methods can be added to the Flight class:
* **class** **Flight**:
* *# assume same \_\_init\_\_ method*
* **def** **print\_info**(self):
* **print**(f"Flight origin: {self.origin}")
* **print**(f"Flight destination: {self.destination}")
* **print**(f"Flight duration: {self.duration}")
* **def** **main**():
* f1 **=** Flight(origin**=**"New York", destination**=**"Paris", duration**=**540)
* f1.print\_info()
  + Now, this functionality of printing out flight info can be used with any flight object that might be created. Each time, self refers to the object that the method is being called on. In this example, that’s f1.
* Methods can also take additional arguments and modify properties.
* **def** **delay**(self, amount):
* self.duration **+=** amount
  + Note that writing methods like delay and print\_info, as well just the idea of Flight class in general, allow for abstraction. The Flight class and all of its methods can be used in a logical and easily understood way without needing to know or even understand how Flight may be implemented.
* Given a simple Passenger class…
* **class** **Passenger**:
* **def** **\_\_init\_\_**(self, name):
* self.name **=** name
* A more complex Flight class can be implemented.
* **class** **Flight**:
* counter **=** 1
* **def** **\_\_init\_\_**(self, origin, destination, duration):
* *# Keep track of id number.*
* self.id **=** Flight.counter
* Flight.counter **+=** 1
* *# Keep track of passengers.*
* self.passengers **=** []
* *# Details about flight.*
* self.origin **=** origin
* self.destination **=** destination
* self.duration **=** duration
* **def** **print\_info**(self):
* **print**(f"Flight origin: {self.origin}")
* **print**(f"Flight destination: {self.destination}")
* **print**(f"Flight duration: {self.duration}")
* **print**()
* **print**("Passengers:")
* **for** passenger **in** self.passengers:
* **print**(f"{passenger.name}")
* **def** **add\_passenger**(self, p):
* self.passengers.append(p)
* p.flight\_id **=** self.id
  + Note that counter is defined outside of the \_\_init\_\_ function and is not specific to individual flights (it’s not defined as self.counter. This means that all flight objects can see this same counter variable, which allows for the implementation the id property shown here. Similar to the SQL database which had an auto-incrementing id column, the id property of flights will automatically increment as new flight objects are created.
  + The passengers property of Flights is going to be a list of Passenger objects.
  + In add\_passenger, p.flight\_id is created, because flight\_id is not defined in the Passenger class’s \_\_init\_\_.
* Here’s how the more advanced Flight class could be used:
* *# Create flight.*
* f1 **=** Flight(origin**=**"New York", destination**=**"Paris", duration**=**540)
* *# Create passengers.*
* alice **=** Passenger(name**=**"Alice")
* bob **=** Passenger(name**=**"Bob")
* *# Add passengers.*
* f1.add\_passenger(alice)
* f1.add\_passenger(bob)
* f1.print\_info()

[**Object Relational Mapping**](https://cs50.harvard.edu/web/2018/notes/4/#object-relational-mapping)

* Object-Relational Mapping, or ORM, allows for the combination of the OOP world of Python and the relational database world of SQL. With ORM, Python classes, methods, and objects become the tools for interacting with SQL databases. To do this, the Flask-SQLAlchemy package will be used.
* The basic setup, inside models.py:
* **from** flask\_sqlalchemy **import** SQLAlchemy
* db **=** SQLAlchemy()
* **class** **Flight**(db.Model):
* \_\_tablename\_\_ **=** "flights"
* id **=** db.Column(db.Integer, primary\_key**=**True)
* origin **=** db.Column(db.String, nullable**=**False)
* destination **=** db.Column(db.String, nullable**=**False)
* duration **=** db.Column(db.Integer, nullable**=**False)
* **class** **Passenger**(db.Model):
* \_\_tablename\_\_ **=** "passengers"
* id **=** db.Column(db.Integer, primary\_key**=**True)
* name **=** db.Column(db.String, nullable**=**False)
* flight\_id **=** db.Column(db.Integer, db.ForeignKey("flights.id"), nullable**=**False)
  + For any table inside of the database, there is one class defined inside models.py.
  + Adding db.Model in parentheses after class names indicates that these classes ‘inherit’ from db.Model. The details of inheritance are unimportant right now; simply, this allows for the class to have some built-in relationship with SQLAlchemy to interact with the database.
  + \_\_tablename\_\_ naturally corresponds with the table name inside the database.
  + Every property is defined as a db.Column, which will become columns in the table. The arguments to db.Column are naturally similar to those use for table creation in SQL.
  + Note that flights.id is marked as a foreign key using the \_\_tablename\_\_ flights, not the class name Flight.
* Now that there’s a defined structure for how the tables should look, they can be created inside a Flask application.
* **import** os
* **from** flask **import** Flask, render\_template, request
* *# Import table definitions.*
* **from** models **import** **\***
* app **=** Flask(\_\_name\_\_)
* *# Tell Flask what SQLAlchemy databas to use.*
* app.config["SQLALCHEMY\_DATABASE\_URI"] **=** os.getenv("DATABASE\_URL")
* app.config["SQLALCHEMY\_TRACK\_MODIFICATIONS"] **=** False
* *# Link the Flask app with the database (no Flask app is actually being run yet).*
* db.init\_app(app)
* **def** **main**():
* *# Create tables based on each table definition in `models`*
* db.create\_all()
* **if** \_\_name\_\_ **==** "\_\_main\_\_":
* *# Allows for command line interaction with Flask application*
* **with** app.app\_context():
* main()

[**Python Versions of SQL Queries**](https://cs50.harvard.edu/web/2018/notes/4/#python-versions-of-sql-queries)

* db.create\_all() is the Python-Flask-SQLAlchemy’s version of the CREATE SQL command.
* SQL’s INSERT…
* INSERT INTO flights
* (origin, destination, duration)
* VALUES ('New York', 'Paris', 540)
* …and Python’s INSERT.
* flight **=** Flight(origin**=**"New York", destination**=**"Paris", duration**=**540)
* db.session.add(flight)
  + SQlAlchemy automatically takes care of SQL transactions with db.session.
* SQL’s SELECT…
* **SELECT** **\*** **FROM** flights;
* **SELECT** **\*** **FROM** flights
* **WHERE** origin **=** 'Paris';
* **SELECT** **\*** **FROM** flights
* **WHERE** origin **=** 'Paris' **LIMIT** 1;
* **SELECT** **COUNT**(**\***) **FROM** flights
* **WHERE** origin **=** 'Paris';
* **SELECT** **\*** **FROM** flights **WHERE** id **=** 28;
* **SELECT** **\*** **FROM** flights
* **ORDER** **BY** origin;
* **SELECT** **\*** **FROM** flights
* **ORDER** **by** origin **DESC**;
* **SELECT** **\*** **FROM** flights
* **WHERE** origin **!=** 'Paris';
* **SELECT** **\*** **FROM** flights
* **WHERE** origin **LIKE** '%a%';
* **SELECT** **\*** **FROM** flights
* **WHERE** origin **IN** ('Tokyo', 'Paris');
* **SELECT** **\*** **FROm** flights
* **WHERE** origin **=** "Paris"
* **AND** duration **>** 500;
* **SELECT** **\*** **FROm** flights
* **WHERE** origin **=** "Paris"
* **AND** duration **>** 500;
* **SELECT** **\*** **FROM** flights **JOIN** passengers
* **ON** flights.id **=** passengers.flight\_id;
* …and Python’s SELECT:
* Flight.query.all()
* Flight.query.fliter\_by(origin**=**"Paris").all()
* Flight.query.filter\_by(origin**=**"Paris").first()
* Flight.query.filter\_by(origin**=**"Paris").count()
* Flight.query.get(28)
* Flight.query.order\_by(Flight.origin).all()
* Flight.query.order\_by(Flights.origin.desc()).all()
* Flight.query.filter(Flight.origin **!=** "Paris").all()
* Flight.query.filter(Flight.origin.like("%a%")).all()
* Flight.query.filter(Flight.origin.in\_(["Tokyo", "Paris"])).all()
* Flight.query.filter(and\_(Flight.origin **==** "Paris", Flight.duration **>** 500)).all()
* Flight.query.filter(or\_(Flight.origin **==** "Paris", Flight.duration **>** 500)).all()
* db.session.query(Flight, Passenger).filter(Flight.id **==** Passenger.flight\_id).all()
* SQL’s UPDATE…
* **UPDATE** flights **SET** duration **=** 280
* **WHERE** id **=** 6;
* …and Python’s UPDATE:
* flight **=** Flight.query.get(6)
* flight.duration **=** 280
* SQL’s DELETE…
* **DELETE** **FROM** flights **WHERE** id **=** 28;
* …and Python’s DELETE:
* flight **=** Flight.query.get(28)
* db.ksession.delete(flight)
* Some other miscellaneous SQL commands…
* **COMMIT**;
* …and their Python parallels.
* db.session.commit()
* Before, when importing data from a CSV file, SQL code had to be written directly into the Python file. Now, SQLAlchemy can take care of that behind the scenes.
* **import** csv
* *# Same setup code as before.*
* **def** **main**():
* f **=** open("flights.csv")
* reader **=** csv.reader(f)
* **for** origin, destination, duration **in** reader:
* flight **=** Flight(origin**=**origin, destination**=**destination, duration**=**duration)
* db.session.add(flight)
* **print**(f"Added flight from {origin} to {destination} lasting {duration} minutes.")
* db.session.commit()

[**ORM Integrated into a Web Application**](https://cs50.harvard.edu/web/2018/notes/4/#orm-integrated-into-a-web-application)

* Putting it all together, here’s the same web application from the end of Lecture 3, using SQLAlchemy. Note that there are no raw SQL commands. The power of ORM, classes, and objects is used to insert and select from the database.
* **from** flask **import** Flask, render\_template, request
* **from** models **import** **\***
* app **=** Flask(\_\_name\_\_)
* app.config["SQLALCHEMY\_DATABASE\_URI"] **=** os.getenv("DATABASE\_URL")
* app.config["SQLALCHEMY\_TRACK\_MODIFICATIONS"] **=** False
* db.init\_app(app)
* **@**app.route("/")
* **def** **index**():
* flights **=** Flight.query.all()
* **return** render\_template("index.html", flights**=**flights)
* **@**app.route("/book", methods**=**["POST"])
* **def** **book**():
* """Book a flight."""
* *# Get form information.*
* name **=** request.form.get("name")
* **try**:
* flight\_id **=** int(request.form.get("flight\_id"))
* **except** ValueError:
* **return** render\_template("error.html", message**=**"Invalid flight number.")
* *# Make sure the flight exists.*
* flight **=** Flight.query.get(flight\_id)
* **if** flight **is** None:
* **return** render\_template("error.html", message**=**"No such flight with that id.")
* *# Add passenger.*
* passenger **=** Passenger(name**=**name, flight\_id**=**flight\_id)
* db.session.add(passenger)
* db.session.commit()
* **return** render\_template("success.html")
* **@**app.route("/flights")
* **def** **flights**():
* """List all flights."""
* flights **=** Flight.query.all()
* **return** render\_template("flights.html", flights**=**flights)
* **@**app.route("/flights/<int:flight\_id>")
* **def** **flight**(flight\_id):
* """List details about a single flight."""
* *# Make sure flight exists.*
* flight **=** Flight.query.get(flight\_id)
* **if** flight **is** None:
* **return** render\_template("error.html", message**=**"No such flight.")
* *# Get all passengers.*
* passengers **=** Passenger.query.filter\_by(flight\_id**=**flight\_id).all()
* **return** render\_template("flight.html", flight**=**flight, passengers**=**passengers)
* Because classes are flexible, whatever additional functionality the app may need can be built into classes. Adding passengers, for example, can be defined as a method in the Flight class (in models.py).
* **def** **add\_passenger**(self, name):
* p **=** Passenger(name**=**name, flight\_id**=**self.id)
* db.session.add(p)
* db.session.commit()
* Now, after verifying that the flight exists, the all that is needed in the book function of application.py is the following:
* flight.add\_passenger(name)
  + Now, there is on direct creation of passengers in the application. It’s all handled by the Flight class.

[**Relationships**](https://cs50.harvard.edu/web/2018/notes/4/#relationships)

* One more powerful feature of ORMs is the idea of relationships. SQLAlchemy relationships are an easy way to take one table and relate it to another table, such that the each can refer to the other. A relationship is set up with a single line, which in this case would be added to the definition of the Flight class.
* passengers **=** db.relationship("Passenger", backref**=**"flight", lazy**=**True)
  + passengers is not a column, but rather just a relationship. Given a flight object, the passengers property can be used to extract all the passenger info for that flight.
  + backref creates a relationship in the opposite direction, from Flight to Passenger.
  + lazy indicates that the information should be fetched only when it’s asked for.k
* With these relationships set up, the code in application.py’s flight function to list get all passengers is extremely simplified.
* passengers **=** flight.passengers
* Once again, SQL’s SELECT…
* **SELECT** **\*** **FROM** passengers
* **WHERE** flight\_id **=** 1
* **SELECT** **\*** **FROM** flights **JOIN** passengers
* **ON** flights.id **=** passengers.flight\_id
* **WHERE** passengers.name **=** 'Alice';
* …and Python’s relationship-powered SELECT.
* Flight.query.get(1).passengers
* Passenger.query.filter\_by(name**=**"Alice").first().flight

[**APIs**](https://cs50.harvard.edu/web/2018/notes/4/#apis)

* An Application Programming Interface, or API, is a protocol for communication between different web applications or different components of the same application. These different components will want to share information with each other or perform actions on other spaces, and APIs allow for this interaction. It is useful, then, to have a standard language for how this communication will occur.

[**JSON**](https://cs50.harvard.edu/web/2018/notes/4/#json)

* One such language is JavaScript Object Notation (JSON), which is a simple way of representing information in human- and computer-readable way so that it can be passed between parts of web application.
* Some example JSON:
* {
* **"origin"** : {
* **"city"**: "Tokyo",
* **"code"**: "HND"
* },
* **"destination"**: {
* **"city"**: "Shanghai",
* **"code"**: "PVG"
* },
* **"duration"** : 185,
* **"passengers"** : ["Alice", "Bob"]
* }
  + The curly braces enclose a JSON object.
  + The contents of the JSON object are divided into key-value pairs.
  + origin and duration are themselves JSON objects, which are nested in a hierarchical structure.
  + passengers shows how lists can be values.
* Often times, the interaction between two APIs happens through the URL, which specifics which particular information that should be accessed. Some different levels might be:
* /flights/
* /flights/28/
* /flights/28/passengers/
* /flights/28/passengers/6/

[**HTTP Methods**](https://cs50.harvard.edu/web/2018/notes/4/#http-methods)

* Often times, there are different ways an API can be used. For example, one might get information about a passenger, register a new passenger, or change registration information for a flight.
* The HTTP request method will correspond to the type of action that should be performed. This is simply a convention that many APIs follow. Some HTTP methods include:
  + GET : retrieve a resource
  + POST : create a new resource
  + PUT : replace a resource
  + PATCH : update a resource
  + DELETE : delete a resource
* The Python Requests library allows for all these different HTTP methods to be used.
* **import** requests
* **def** **main**():
* res **=** requests.get("https://www.google.com/")
* **print**(res.text)
* \* `res` (response) is the HTTP response that comes from submitting, in this case, a `GET` request to a URL. All the following are also valid:
* \* `requests.post(url)`
* \* `requests.put(url)`
* \* `requests.patch(url)`
* \* `requests.delete(url)`
* \* `res.text` is the HTML content of the page that is returned from the request.

[**An Example API**](https://cs50.harvard.edu/web/2018/notes/4/#an-example-api)

* To demonstrate the potential for these requests, [Fixer](https://fixer.io/), a foreign exchange rate API, will be used in the following examples.
* Accessing the API at the URL http://data.fixer.io/api/latest?access\_key=apikey&base=EUR&symbols=USD’ returns the following JSON:
* {
* **"success"**: **true**,
* **"timestamp"**: 1519296206,
* **"base"**: "EUR",
* **"date"**: "2018-07-11",
* **"rates"**: {
* **"USD"**: 1.177482
* }
* }
* This API can be accessed in Python using the Requests library.
* res **=** requests.get("http://data.fixer.io/api/latest?access\_key=apikey&base=EUR&symbols=USD")
* **if** res.status\_code **!=** 200:
* **raise** Exception("ERROR: API request unsuccessful.")
* data **=** res.json()
* **print**(data)
* \* Checking the status code of the HTTP response ensures that the API returned what is expected by the application (a JSON formatted like the one above). As an aside, here are some common HTTP status codes. Generally, a leading 2 indicates a successful response, while a leading 4 indicates a failed request.
* \* `200 OK`
* \* `201 Created`
* \* `400 Bad Request`
* \* `403 Forbidden`
* \* `404 Not Found`
* \* `405 Method Not Allowed`
* \* `422 Unprocessable Entity`
* \* `res.json()` simply extracts the JSON response and puts into the Python variable `data`.
* The previous returned the entire, raw JSON returned by the API. Since the format of the JSON is consistent and known, the most relevant information can be extracted and displayed.
* rate **=** data["rates"]["USD"]
* **print**(f"1 EUR is equal to {rate} USD")
* For a little more flexibility on what currencies are being converted, user input can be taken like so:
* base **=** input("First Currency: ")
* other **=** input("Second Currency: ")
* res **=** requests.get("http://data.fixer.io/api/latest",
* params**=**{"access\_key": apikey, "base": base, "symbols": other})
  + What parameters should be passed into params (in this case, "access\_key", "base" and "symbols") are defined in the API documentation.

[**Creating an API**](https://cs50.harvard.edu/web/2018/notes/4/#creating-an-api)

* To implement an API for the recurring example of a airline flight manager, all that needs to be done is to define a route that returns a JSON object, just like Fixer does.
* **from** flask **import** Flask, render\_template, jsonify, request
* *# ... other imports, set up code, and routes ...*
* **@**app.route("/api/flights/<int:flight\_id>")
* **def** **flight\_api**(flight\_id):
* """Return details about a single flight."""
* *# Make sure flight exists.*
* flight **=** Flight.query.get(flight\_id)
* **if** flight **is** None:
* **return** jsonify({"error": "Invalid flight\_id"}), 422
* *# Get all passengers.*
* passengers **=** flight.passengers
* names **=** []
* **for** passenger **in** passengers:
* names.append(passenger.name)
* **return** jsonify({
* "origin": flight.origin,
* "destination": flight.destination,
* "duration": flight.duration,
* "passengers": names
* })
* The route URL is clearly marked as an API, and takes any flight ID as a parameter.
* jsonify is a function provided by Flask that takes in a Python dictionary and converts it into JSON.
* If there is no flight found, an HTTP status code (422) is also returned with the JSON to indicate an error has occurred.
* Seen here again is the readability and simplicity of relationships when retrieving passenger information.
* If a valid flight ID was passed as a parameter, then a JSON object with all the flight info and a list of passengers is returned (because no status code is specified, it is set to 200 by default).

[**API Keys**](https://cs50.harvard.edu/web/2018/notes/4/#api-keys)

* With larger APIs, an often-implemented feature is rate limiting. It is undesirable to have users making a large number of requests that might overload the API or make it harder for other users to access it. To restrict access, users must first obtain an API key (a long string) which must be provided with any API request. Keys allow for the tracking of individual users only allowing, for example, 100 requests per hour per user.