# Lab 3: Learning an API

For many projects this semester, we'll provide you with a *module* (a collection of functions) named `project`, in a file named `project.py`. This module will provide functions that will help you complete the project. Today, we'll introduce the `project.py` file you'll use for P3.

When using an unfamiliar module, the first thing you should do is study the module's *API*. API stands for "Application Programming Interface", and refers to a collection of related functions (e.g., those in a module). Understanding the API will involve learning about each function and the parameters it takes. You might also need to learn new *protocols* for using the function; protocols specify the order in which you may call functions.

There are two ways you can learn about an API. First, the person who created the API may have provided written directions, called *documentation*. Second, there are ways to can write code to learn about a collection of functions; this approach is called *inspection*.

Summary of new terms:

- module
- API
- protocol
- documentation
- inspection

## Note on Academic Misconduct

You may do these lab exercises with anybody you like. But be careful! It's very natural to start working on P3 immediately after completing the lab. If you start working with somebody on P3 (after the lab), that person must be your project partner until the next project; you are not allowed to start working on P3 with one person during lab, then finish the project with a different partner. Now may be a good time to review our course policies.

## Setup

Create a `lab3` directory and download `lab.csv` above. Also download these files from the P3 posting to the `lab3` directory:

- `madison.csv`
- `project.py`

Open a terminal and navigate to your `lab3` directory. Run `ls` to make sure your three files are available.

We'll be doing these exercises in interactive mode, so type `python` (or `python3`, if that's what you need to do on your laptop), and hit ENTER.

## Inspecting `__builtins__` and `math`

In interactive mode, try the following examples (only type things after the `>>>`).

```
>>> abs(-4)
4
>>> x = abs(-3)
>>> x
3
```

These two calls invoke the `abs` function because we have parenthesis. What if we don't use parenthesis? Try the following and see what you get:

```
>>> abs
```

```
>>> type(abs)
```

What if we want to read about what `abs` does? Run this:

```
>>> abs.__doc__
```

Or this (compare the result):

```
>>> print(abs.__doc__)
```

We didn't need to import anything to use `abs` because it is part of a special module that is always imported called `__builtins__`. Try running this to see:

```
>>> type(__builtins__)
```

The `dir` function will show you everything that is inside a module, so let's use it to learn about `__builtins__`. Run this:

```
>>> dir(__builtins__)
```

This displays the names of lots of functions we've seen, such as `abs`, `print`, `int`, `input`, and others. You'll see some things that begin and end with `--`. Those are generally things to ignore. Choose one function from the list that you're familiar with and one that is unfamiliar to you, and then use `.__doc__` to read the descriptions for both. For example, you might learn about `max` like this:

```
>>> print(max.__doc__)
max(iterable, *[, default=obj, key=func]) -> value
max(arg1, arg2, *args, *[, key=func]) -> value

With a single iterable argument, return its biggest item. The
default keyword-only argument specifies an object to return if
the provided iterable is empty.
With two or more arguments, return the largest argument.
```

Wow, that mentions a lot of things we haven't learned about yet! As a new Python programmer reading documentation, you'll have to dig through things you don't understand yet to find bits that are useful for you. For example, in this case, the last line tells you everything you need to know: *"With two or more arguments, return the largest argument."*

Let's give it a try:

```
max(-1000, 99, 50, 60)
```

Let's see what's in the math module now:

```
>>> import math
>>> dir(math)
```

Let's see what the `log` function does:

```
>>> print(math.log.__doc__)
```

As a convention, documentation that displays parameters in brackets (like `[base=math.e]`) mean that the parameter is optional. Let's try calling the `log` function different ways:

1. `math.log(10000, 10)` (positional arguments)
2. `math.log(math.e ** 3)` (positional argument and default argument)
3. `math.log(x=10000, base=10)` (keyword arguments)

Note that the last command fails with `TypeError: log() takes no keyword arguments`. Not every function you encounter will support keyword arguments, unfortunately.

What happens if you run this?

```
>>> log(10000, 10)
```

It doesn't work because we've imported math with `import math`. Try this style instead, then repeat that call:

```
>>> from math import log
>>> log(10000, 10)
```

We'll still need to use `math.sqrt(4)` instead of `sqrt(4)`, though, unless we specifically import `sqrt` like we did for `log`. Or, we can import everything in math at once:

```
>>> from math import *
```

Try some other mathematical functions to verify you don't need to start the calls with `math.`.

## Inspecting `project`

Let's check out the `project.py` API you'll use for P3:

```
>>> import project
>>> dir(project)
['__DictReader', '__agency_to_id', '__builtins__', '__cached__', '__data',
'__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__',
'dump', 'get_id', 'get_spending', 'init']
```

We see there are three functions here (ignoring the things beginning with two underscores):

- dump
- get_id
- get_spending
- init

What does `dump` do?

```
>>> print(project.dump.__doc__)
```

Let's try calling it then:

```
>>> project.dump()
```

You'll get an error:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/trh/git_co/cs301-projects/spring19/lab3/project.py", line 36, in
dump
    raise Exception("you did not call init first")
Exception: you did not call init first
```

Let's figure out what init is, then:

```
>>> print(project.init.__doc__)
```

Looks like it wants us to do this:

```
project.init("madison.csv")
```

After calling init, let's try calling `project.dump()` again. It should show this now:

```
fire    [ID 22]
  2015: $49.737579 MILLION
  2016: $51.968340 MILLION
  2017: $53.144053 MILLION
  2018: $55.215007 MILLION

library [ID 33]
  2015: $16.965434 MILLION
  2016: $18.125521 MILLION
  2017: $19.136348 MILLION
  2018: $19.845066 MILLION

parks   [ID 44]
  2015: $18.371421 MILLION
  2016: $19.159243 MILLION
  2017: $19.316837 MILLION
  2018: $19.760710 MILLION

police  [ID 11]
  2015: $68.063469 MILLION
  2016: $71.325756 MILLION
  2017: $73.247948 MILLION
  2018: $77.875535 MILLION

streets [ID 55]
  2015: $25.368880 MILLION
  2016: $28.228622 MILLION
  2017: $26.655754 MILLION
  2018: $27.798934 MILLION
```

This is actual spending data for five large agencies in the City of Madison over the last four years.

Why do we need to call `init` before `dump` and other functions? Because `init` loads data from a CSV file (CSV files are like simple spreadsheets), and you might want to also use other CSV files. For example, try this to see some smaller agencies that we saved in `lab.csv`:

```
>>> project.init("lab.csv")
WARNING!  Opening a path other than madison.csv.  That's fine for testing your
code yourself, but madison.csv will be the only file around when we test your
code for grading.
>>> project.dump()
attorney [ID 4]
  2015: $2.703978 MILLION
  2016: $2.775633 MILLION
  2017: $2.989084 MILLION
  2018: $2.993189 MILLION

clerk   [ID 5]
  2015: $1.292095 MILLION
  2016: $2.394929 MILLION
```

```
  2017: $2.266173 MILLION
  2018: $2.223049 MILLION

mayor   [ID 3]
  2015: $1.440244 MILLION
  2016: $1.362939 MILLION
  2017: $1.522648 MILLION
  2018: $1.535043 MILLION
```

What about the `get_id` and `get_spending`? Print the documentation for those too, as you have for other functions (i.e., using `.__doc__`).

As you may have noticed, each department has an ID and a name. `get_spending` looks up spending in a specific year, given an ID. `get_id` looks up an ID given a name. Try a few uses:

- `project.get_id("mayor")` (looks up ID of major agency, which should be 3)
- `project.get_spending(3, 2015)` (looks up spending of agency 3 in 2015, which should be 1.44024423)
- `project.get_spending(project.get_id("mayor"), 2015)` (looks up spending of mayor agency in 2015; mayor agency has ID 3)
- `project.get_spending(project.get_id("mayor"))` (looks up spending of mayor agency in 2018, the default year argument)

Try switching back to the madison.csv dataset (with `project.init("madison.csv")`) and see if you can lookup spending on parks in 2018.

You should also experiment with the three ways to initialize parameters:

- `project.get_spending(11, 2018)` (positional argument for year)
- `project.get_spending(11, year=2018)` (positional argument for year)
- `project.get_spending(11)` (default argument for year)

Finally, you should try some commands that will fail and note the kinds of errors produced:

- `project.get_id("bad")`
- `project.get_spending(0, 2018)`
- `project.get_spending(11, 2019)`
- `project.get_spending(11, 2019, year=2019)`
- `project.init("BAD.csv")`
- `project.init(301)`
- `project.init()`
- `project.dump(True)`

# Project 3

Great, now you're reading to start P3! All the things you've been doing here in interactive mode will work in your notebook as well. Remember to only work with at most one partner on P3 from this point on. Have fun!