# ISTA 350 - Spring 2020
# Lab 5: Classes – SQL, Magic Methods, Operator Overloading, & Decorators

17 February 2020

## Databases

An organized collection of data in which a computer program can easily access the data.

The data in a Database is stored in tables

| customer_id INTEGER PRIMARY KEY | first TEXT | last TEXT | car_year INTEGER | make TEXT | model TEXT | price INTEGER |
|---|---|---|---|---|---|---|
| **1** | 'John' | 'Smith' | 2016 | 'Ford' | 'Mustang' | 36395 |
| **2** | 'Clark' | 'Kent' | 2016 | 'Chevrolet' | 'Camaro' | 37295 |

## Structured Query Language (SQL)

This is how a programmer can obtain and insert information to a database. We will be using sqlite3 in this course. This is a module that you will import in your assignments.

**Making a Database and one Table:**

```
>>> import sqlite3    # Don't forget to import the module
>>>
>>> car_db = sqlite3.connect('sold_cars.db')
>>>
>>> car_db.row_factory = sqlite3.Row # Use to gain easy access to the
                                     # fields of a row
>>>
>>> car_db.execute('CREATE TABLE sold_cars (customer_id INTEGER
PRIMARY KEY, first TEXT, last TEXT, car_year INTEGER, make TEXT,
model TEXT, price INTEGER)')
<sqlite3.Cursor object at 0x101c5a7a0>
>>> car_db.commit() # Make the changes persistent (Save changes)
```

**Note:** `row_factory = sqlite3.Row` allows for index-based and case-insensitive name-based access to columns.

**Adding a Row of Data:**

```
>>> db.execute('INSERT INTO sold_cars (customer_id, first, last,
car_year, make, model, price) VALUES (?,?,?,?,?,?,?);', (1, 'John',
'Smith', 2016, 'Ford', 'Mustang', 36395))
<sqlite3.Cursor object at 0x101c5a3b0>
>>> car_db.commit()
```

**Retrieving a Row:**

Use `.fetchone()` to retrieve a single row or `.fetchall()` to retrieve to get a list of all the rows. Please not this is based on the query in the `.execute()` call.

**Using `fetchone()` method:**

```
>>> cursor = car_db.execute('SELECT * FROM sold_cars;') # Create
cursor to access data within the database
>>> row = cursor.fetchone()
>>> row[0]
>>> 1
```

**Using `fetchall()` method:**

Please note that `fetchall()` will fetch all remaining rows in the database. If there are no remaining rows to be fetched then it will return an empty list.

**This example assumes that we did not execute the block of code above.**

```
>>> car_db.execute('INSERT INTO sold_cars (customer_id, first, last,
car_year, make, model, price) VALUES (?,?,?,?,?,?,?);', (2, 'Clark',
'Kent', 2016, 'Chevrolet', 'Camaro', 37295)) #adding more data to db
<sqlite3.Cursor object at 0x101c5a3b0>
>>> cursor = car_db.execute('SELECT * FROM sold_cars;')
>>> rows = cursor.fetchall()
>>> len(rows)
>>> 2
>>> rows[0]['first']
'John'
>>> for row in rows:
...     print(tuple(row)) #Use tuple to represent the data.
('1', 'John', 'Smith', 2016, 'Ford', 'Mustang', 36395)
('2', 'Clark', 'Kent', 2016, 'Chevrolet', 'Camaro', 37295)
```

**Deleting a Row:**

```
>>> car_db.execute('DELETE FROM sold_cars WHERE customer_id =
?;',(2,))
<sqlite3.Cursor object at 0x101c5a3b0>
>>> car_db.commit()
```

## Magic Methods

Python's magic methods, documented as special methods, are methods that you can define in your classes that allow objects to behave like built-in types. Unlike normal class methods, that must be explicitly called to invoke them, magic methods are invoked by Python under special conditions or when a specific syntax is used.

You have already seen and defined a few magic methods, which start and end with "__" (two underscores). In this course you will generally only be dealing with arithmetic `(+, -, *, /)` and/or relational `(<, >, <=, >=)` operator magic methods.

For this example, we will alter Python's `__call__` method. This method is invoked when the instance of the class is called.

In the `AlwaysSmiling` class, the `__call__` method has been altered to print a `":)"` when the class instance is called.

```
class AlwaysSmiling:

    def __call__(self):
        print(":)")
```

```
>>> smile = AlwaysSmiling()
>>> smile()
>>> :)
```

Below is a useful link:
https://docs.python.org/3/reference/datamodel.html#special-method-names

## Operator Overloading:

You can use operator overloading on Python magic methods to overwrite how the operators work. This allows you to extend arithmetic (`**, *, /, +, -, \`) and relational (`<, <=, >, >=`) operators and some built-in function to work on instances of the classes that you define. You determine the behavior of your objects when these operators are used on them.

In the example below we are overloading the addition operator. When we add two instances of the `PrimeColor` class another color (string) should be returned. Note that the only colors you can add together are blue, red, and yellow.

```
class PrimeColor:

    def __init__(self):
        while True:
            pc = input("select color: blue, red, or yellow: ").lower()
            if pc not in ["blue", "red", "yellow"]:
                print("Not valid, try again")
            else:
                self.name = pc
                break

    def __repr__(self):
        return self.name

    def __add__(self, other):
        col1 = self.name
        col2 = other.name

        if col1 == col2:
            return col1
        elif col1 == 'blue' or col2 == 'blue':
            if col2 == 'red' or col1 == 'red':
                return 'purple'
            elif col2 == 'yellow' or col1 == 'yellow':
                return 'green'
        elif col1 == 'red' or col2 == 'red':
            if col1 == 'yellow' or col2 == 'yellow':
                return 'orange'
```

```
>>> blue = PrimeColor()
>>> select color: blue, red, or yellow: BLue
>>> red = PrimeColor()
>>> select color: blue, red, or yellow: RED
>>> blue + red
>>> purple
```

## Decorators:

In general, a decorator is a function that wraps over another function in order to modify the behavior of the function without actually modifying the code in the function itself.

In Python, the operator for declaring a decorator is `'@'`. Class and function decorators must be declared immediately prior to its definition. Once a decorator is declared you add functionality as well as change the behavior of the declared class or function.

### Static Methods

In this course we will be declaring static methods. Static methods are used to include functions in a class that logically belong to the class but do not need to be invoked on induvial instance of the class. Meaning that static methods do not utilize the `self` instance parameter so you can call the method without instantiating the class first.

```
class UofAStudent:
    def __init__(self, name, year, major, minor = None):
        self.name = name
        self.year = year
        self.major = major
        self.minor = minor

    def enroll(self):
        print("Greetings " + self.name)
        option = input("Would you like to enroll to the U of A? ")
        if option.lower() == 'yes':
            print("Congratulations you made the right choice!")
        else:
            print("Your loss!")

    @staticmethod
    def uOfa_slogan():
        print("Bear Down!")
```

```
>>> UofAStudent.uOfa_slogan()
>>> Bear Down!
```

Note: You didn't need to instantiate a `UofAStudent` object before calling the `uOfa_slogan` method.

### Practice

Download and complete each to do item throughout lab4.py.