CSCI-UA-0002

Intro to Computer Programming (No Prior Experience)

Module 11: Object Oriented Programming

Professor Emily Zhao

Section 008
T/R 12:30-1:45PM

Section 012
T/R 4:55-6:10PM

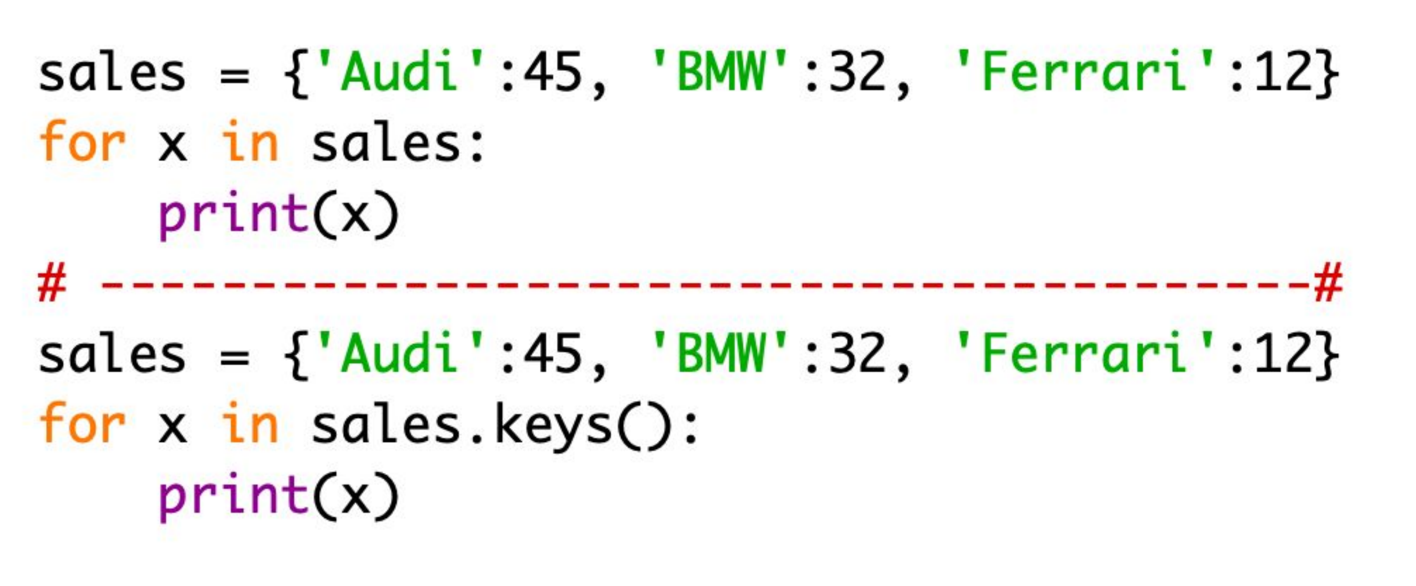
Agenda

- Quick Review of Dictionaries
- Intro to Object Oriented Programming
- Practice Problems

Review

Review: True or False

1) Lists can be keys in dictionaries. > False
2) The following two programs will have the same output. > True



```python
sales = {'Audi':45, 'BMW':32, 'Ferrari':12}
for x in sales:
    print(x)
# -----------------------------------------#
sales = {'Audi':45, 'BMW':32, 'Ferrari':12}
for x in sales.keys():
    print(x)
```

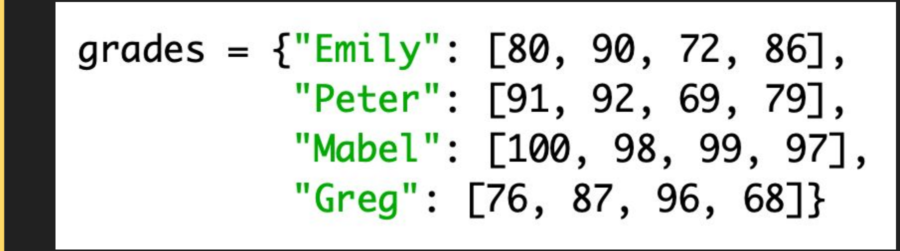
Review

Given the following dictionary:

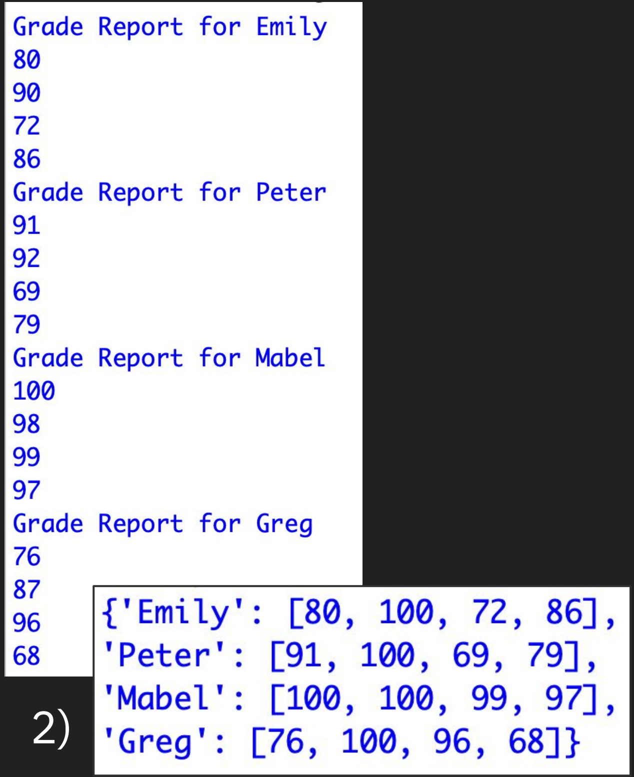
1) Print out a grade report for each student.

2) Change everyone's second grade to 100

3) Drop their lowest grade



```
grades = {"Emily": [80, 90, 72, 86],
          "Peter": [91, 92, 69, 79],
          "Mabel": [100, 98, 99, 97],
          "Greg": [76, 87, 96, 68]}
```



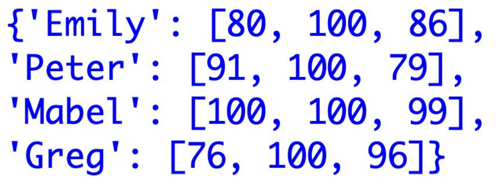
1)

```
Grade Report for Emily
80
90
72
86
Grade Report for Peter
91
92
69
79
Grade Report for Mabel
100
98
99
97
Grade Report for Greg
76
87
96
68
```

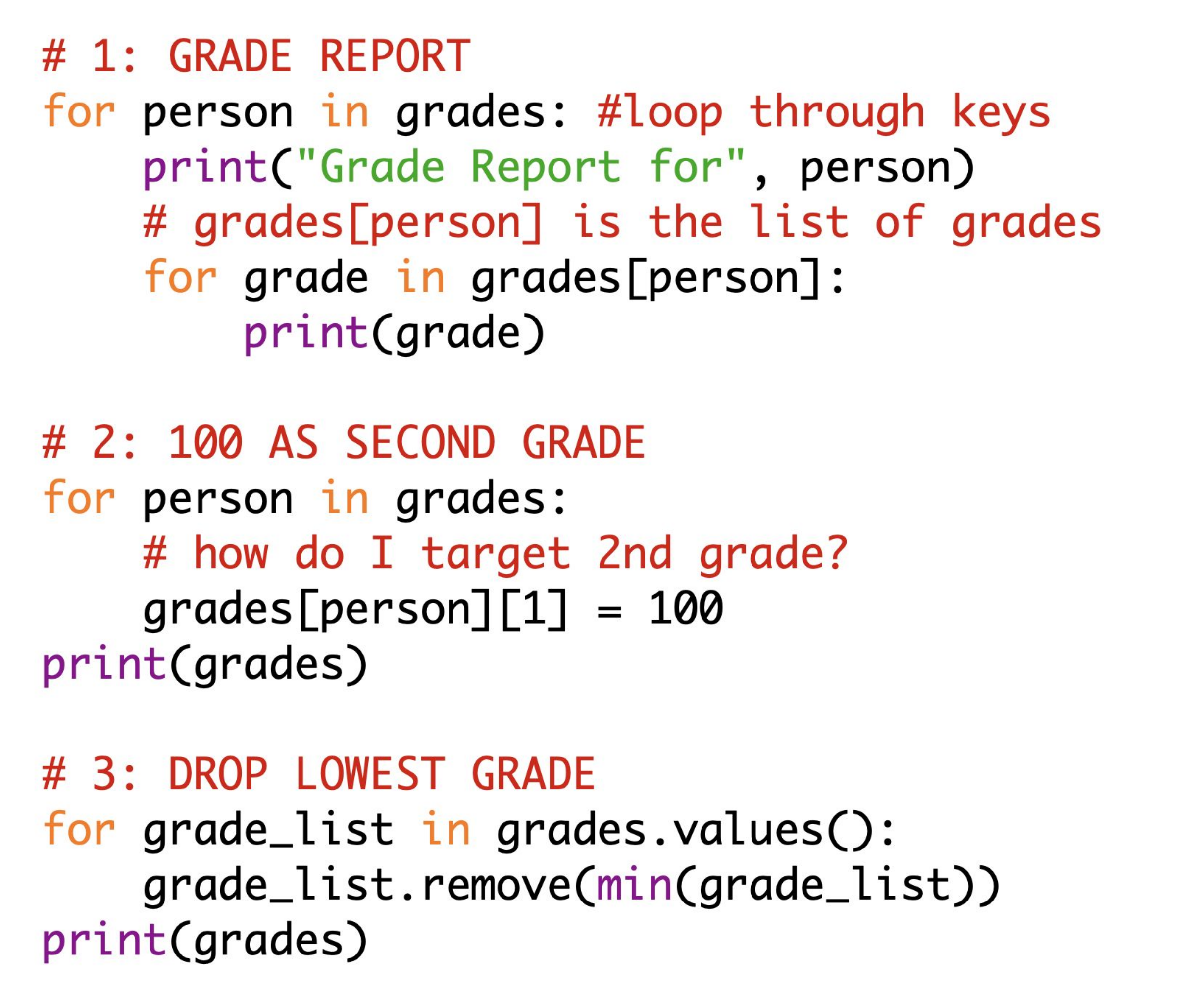
2)

```
{'Emily': [80, 100, 72, 86],
'Peter': [91, 100, 69, 79],
'Mabel': [100, 100, 99, 97],
'Greg': [76, 100, 96, 68]}
```

3)



```
{'Emily': [80, 100, 86],
'Peter': [91, 100, 79],
'Mabel': [100, 100, 99],
'Greg': [76, 100, 96]}
```



```python
# 1: GRADE REPORT
for person in grades: #loop through keys
    print("Grade Report for", person)
    # grades[person] is the list of grades
    for grade in grades[person]:
        print(grade)

# 2: 100 AS SECOND GRADE
for person in grades:
    # how do I target 2nd grade?
    grades[person][1] = 100
print(grades)

# 3: DROP LOWEST GRADE
for grade_list in grades.values():
    grade_list.remove(min(grade_list))
print(grades)
```

Object Oriented Programming

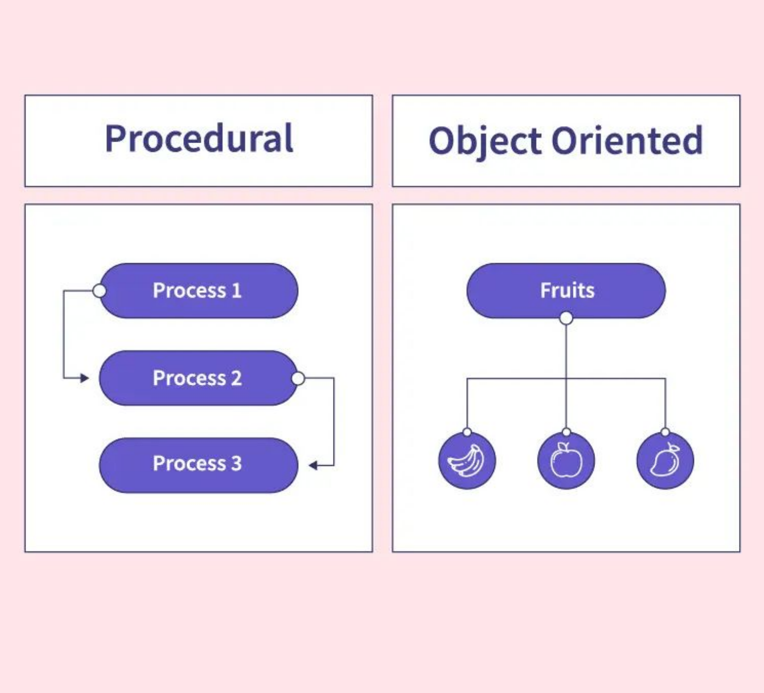
Your Questions

- → When should we use a class?
- → What is instantiation?
  - → Can you go over the constructor? What is `__init__`?
  - → What is `self`? Do you need it? Can it be replaced? Is it special?
- → What is the difference between a function and a method?
- → Is `str` a class?

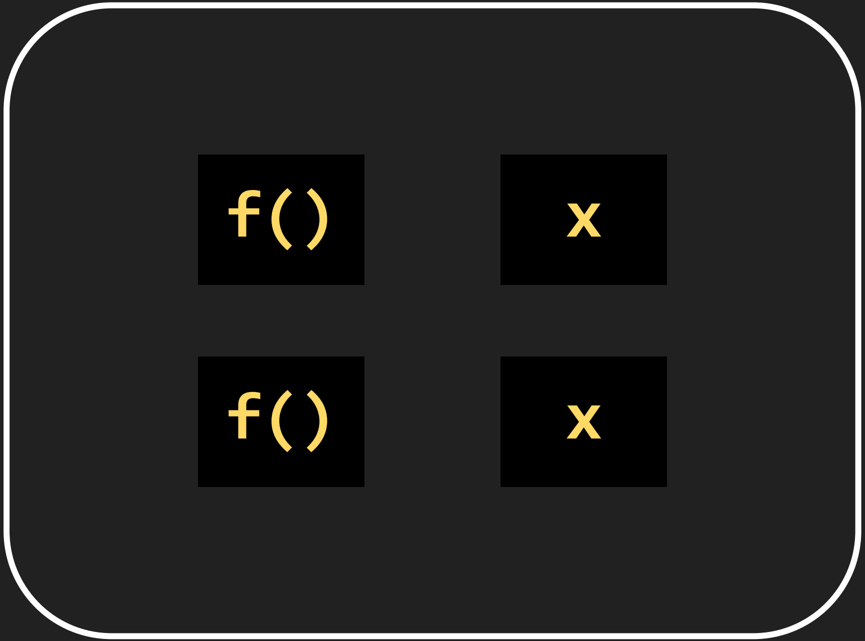
Procedural vs OOP

- Procedural programming is a method of writing software. It is a programming practice centered on the **procedures or actions** that take place in a program.

- Object-oriented programming is centered on **objects**.
  - Objects are created from abstract data types that encapsulate data and functions together.



Procedural



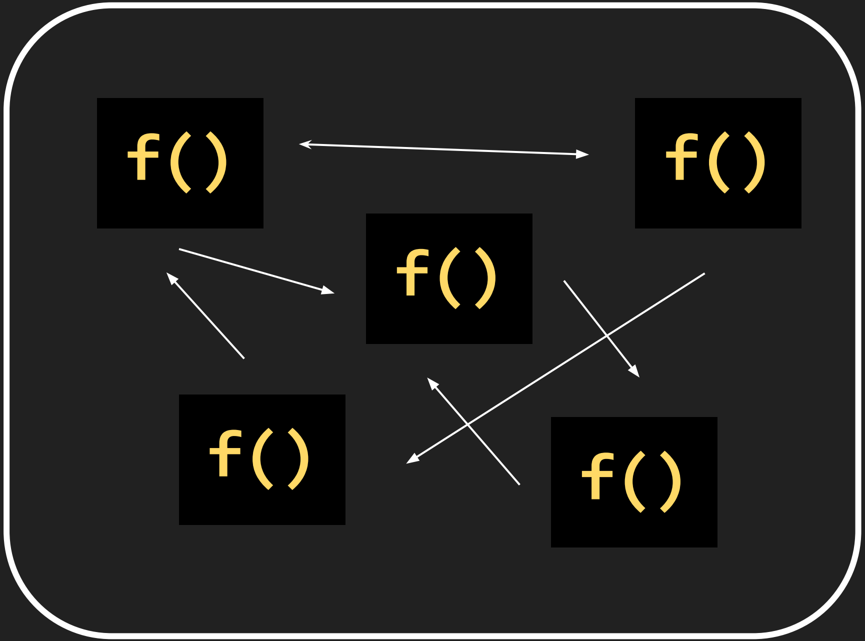
f()

x

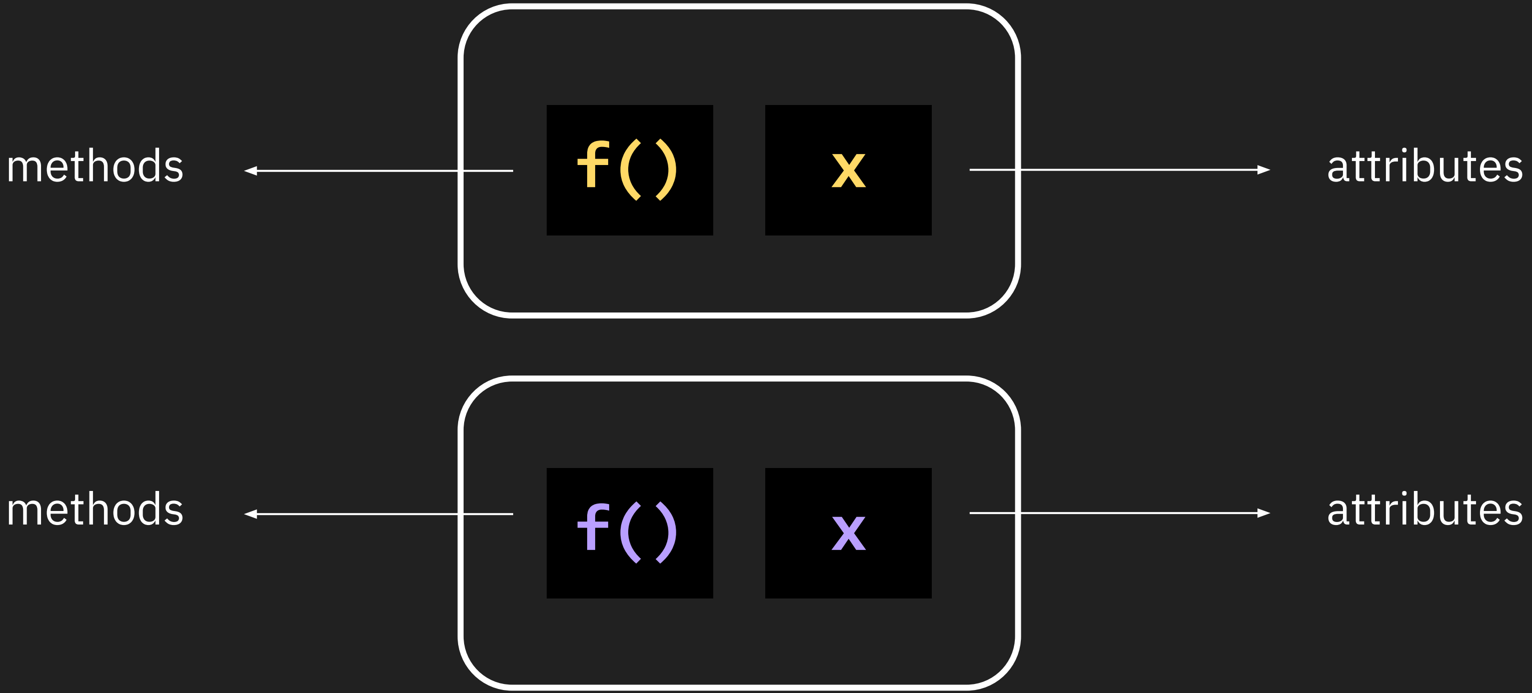
f()

x

Procedural



Object-Oriented



methods

f()

x

attributes

methods

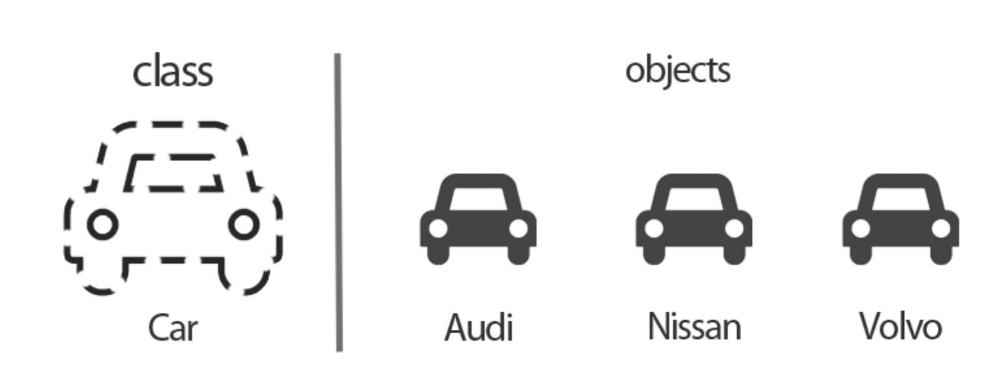
f()

x

attributes

A class is a blueprint

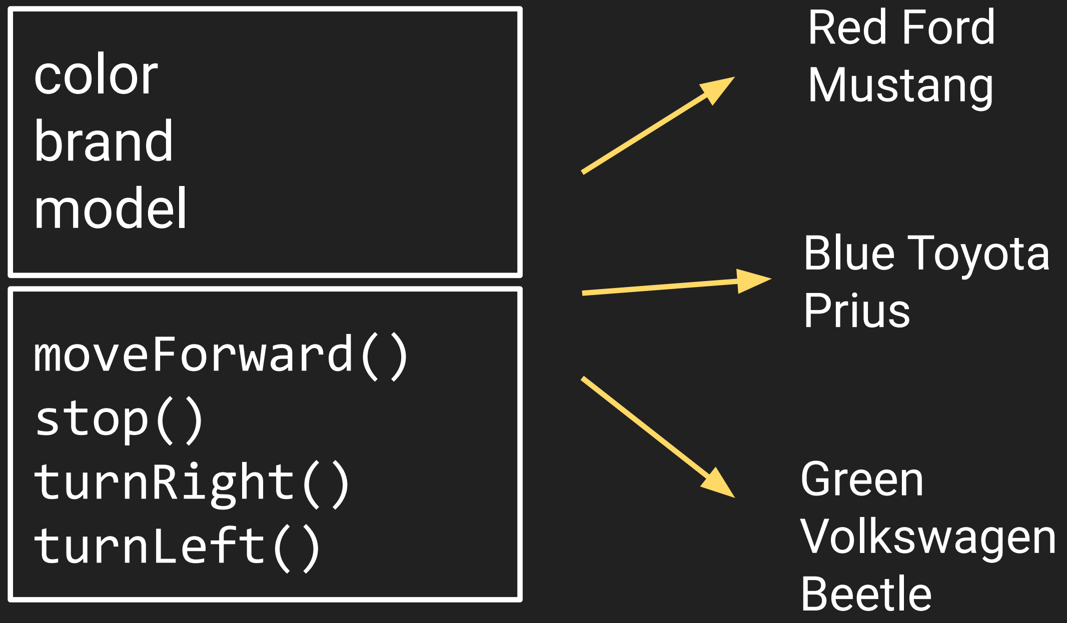
- A class is code that specifies the data attributes and methods for a particular type of object.
  - It is a description of an object’s characteristics.
  - Classes are a blueprint that allow us to make many independent copies of objects that look or behave in similar ways.
- Each object that is created from a class is called an **<u>instance</u>** of the class.



`class Car`

What do all cars have?
(aka attributes)

What do all cars do?
(methods)



color
brand
model

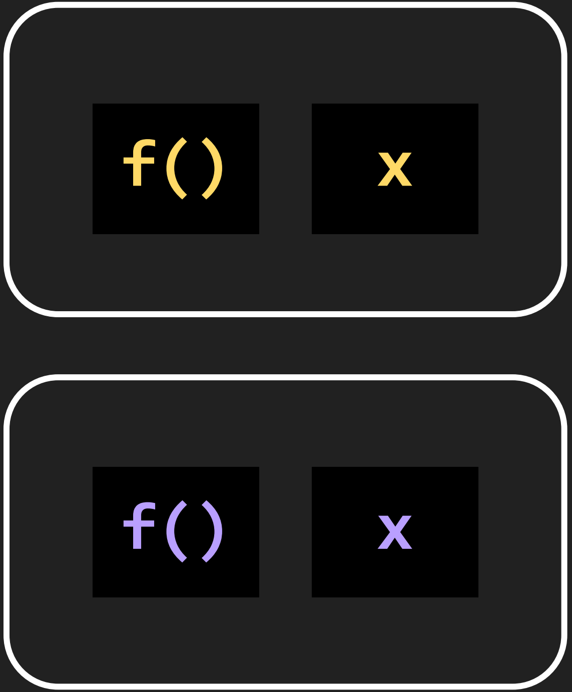
```
moveForward()
stop()
turnRight()
turnLeft()
```

Red Ford
Mustang

Blue Toyota
Prius

Green
Volkswagen
Beetle

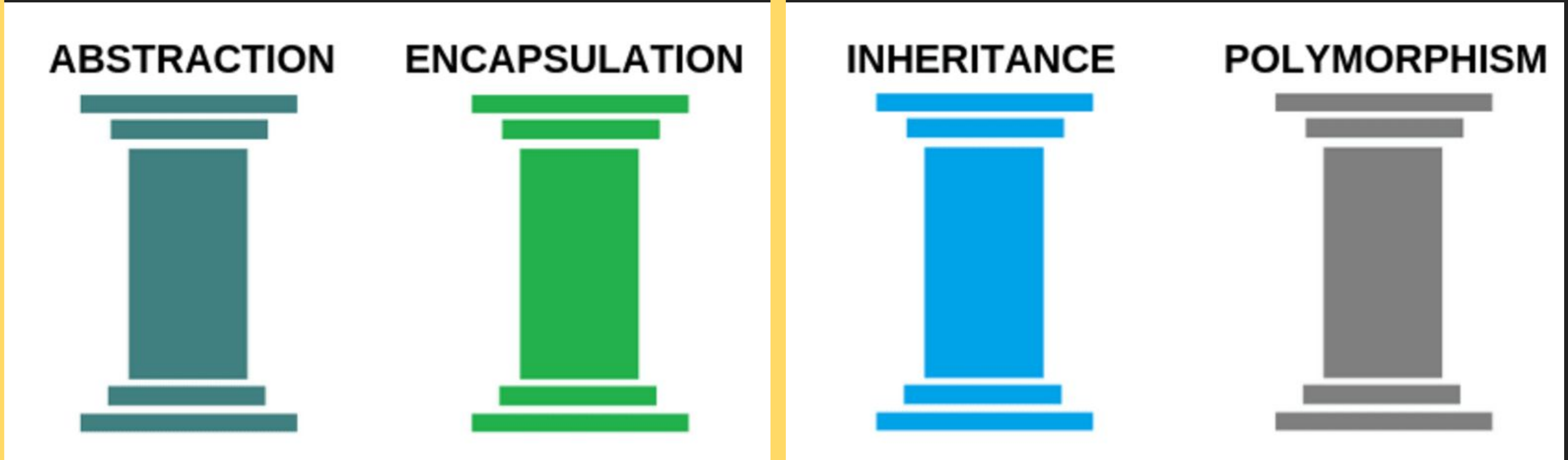
Object-Oriented



The grouping of related functions and variables is called **encapsulation**, one of the fundamental “pillars” of object-oriented programming.

- hides the internal details of an object and restricts access to certain parts
- this makes the implementation details hidden from the outside world

4 Pillars of Object Oriented Programming



grouping of information

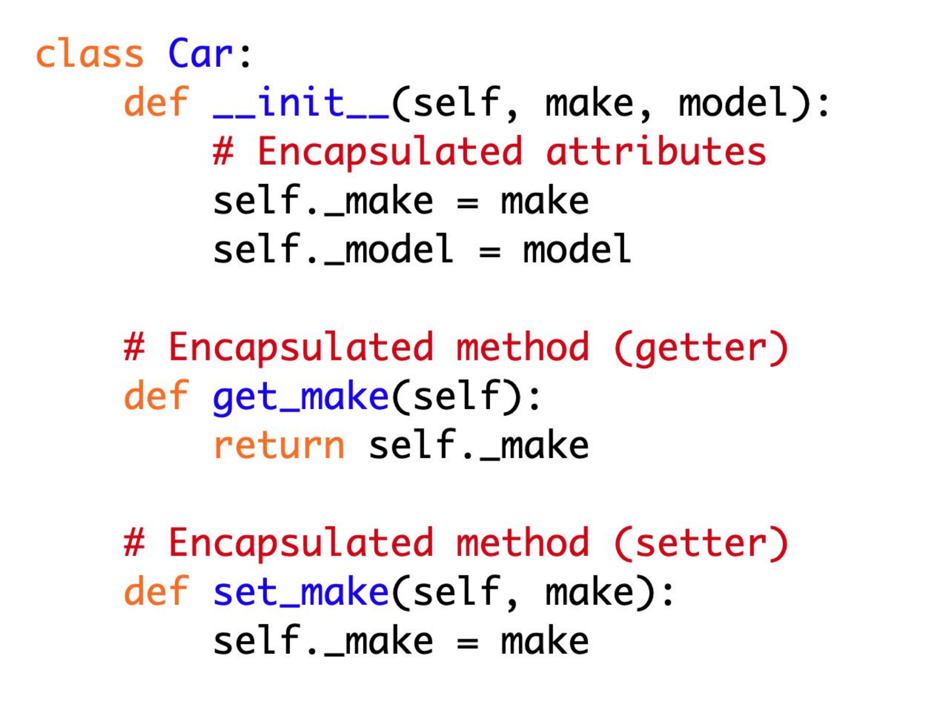
hiding of information

sharing of information

redefining of information

4 Pillars of Object Oriented Programming

hiding of
information



```python
class Car:
    def __init__(self, make, model):
        # Encapsulated attributes
        self._make = make
        self._model = model

    # Encapsulated method (getter)
    def get_make(self):
        return self._make

    # Encapsulated method (setter)
    def set_make(self, make):
        self._make = make
```

Getters and Setters

Getters

- A method that returns a value from a class’s attribute but does not change it is known as an accessor method.

- Accessor methods provide a safe way for code outside the class to retrieve the values of attributes, without exposing the attributes in a way that they could be changed by the code outside the method.

Setters

- A method that stores a value in a data attribute or changes the value of a data attribute in some other way is known as a **mutator method**.

- Mutator methods can control the way that a class’s data attributes are modified. They usually accept a new value as an argument

4 Pillars of Object Oriented Programming

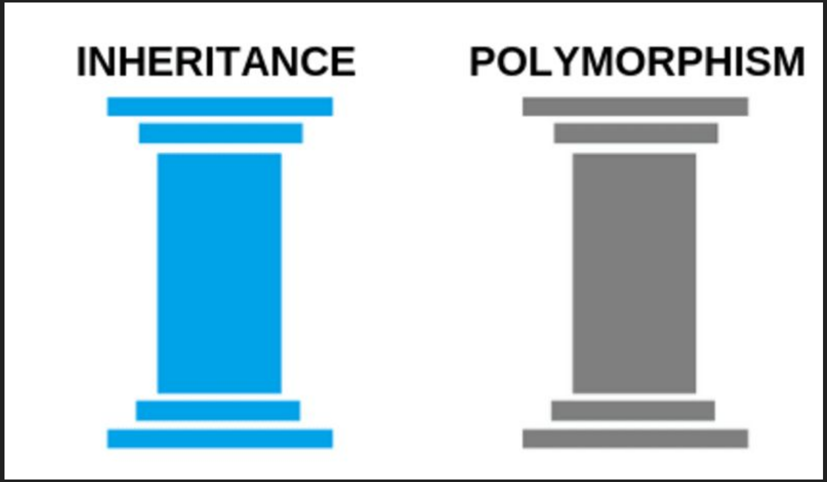
grouping of
information

Make coffee button

versus

Add cold water button
Boil the water button
Add 1 spoon ground coffee button
Clean dirty cups button
etc...

4 Pillars of Object Oriented Programming



sharing of information

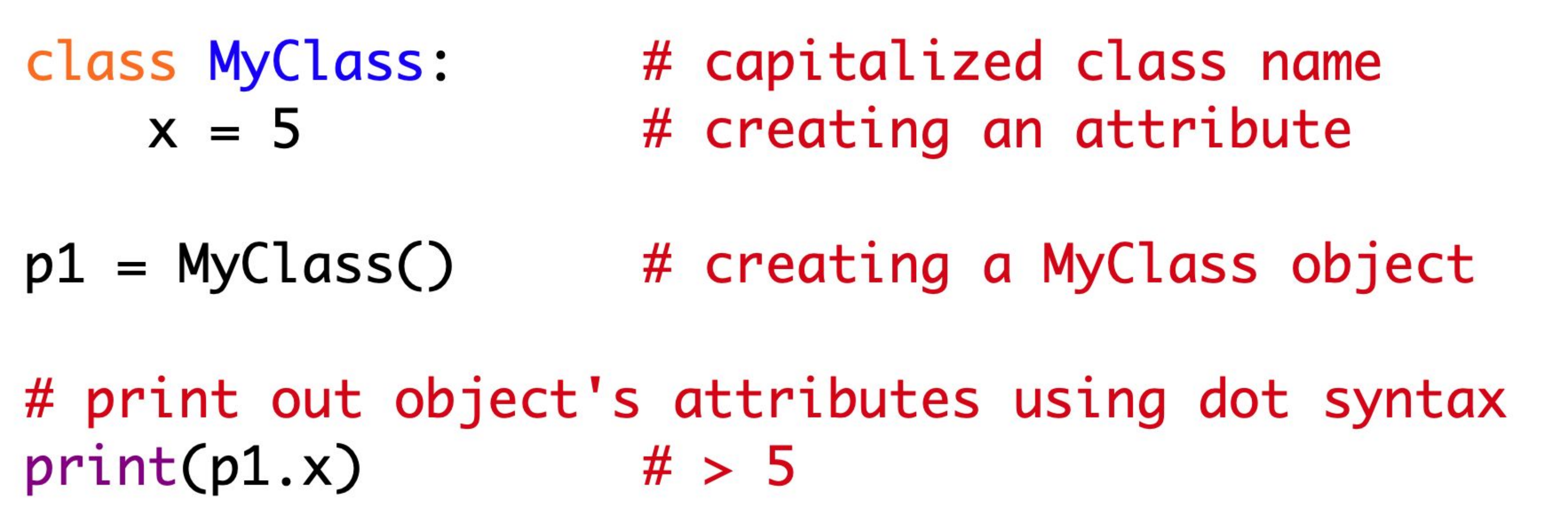
redefining of information

Inheritance
Child classes inherit behavior from parent class and overwrite when necessary (i.e. Shape and its subclass Circle)

Polymorphism **(“many forms”)**
An area method would be polymorphic, meaning that its behavior varies depending on the actual type of the object it is called on.

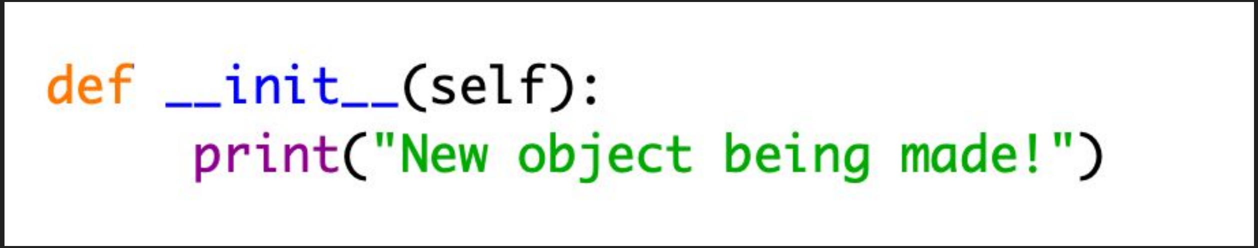
Classes and Objects

How to write a class



```
class MyClass:      # capitalized class name
    x = 5           # creating an attribute

p1 = MyClass()      # creating a MyClass object

# print out object's attributes using dot syntax
print(p1.x)         # > 5
```

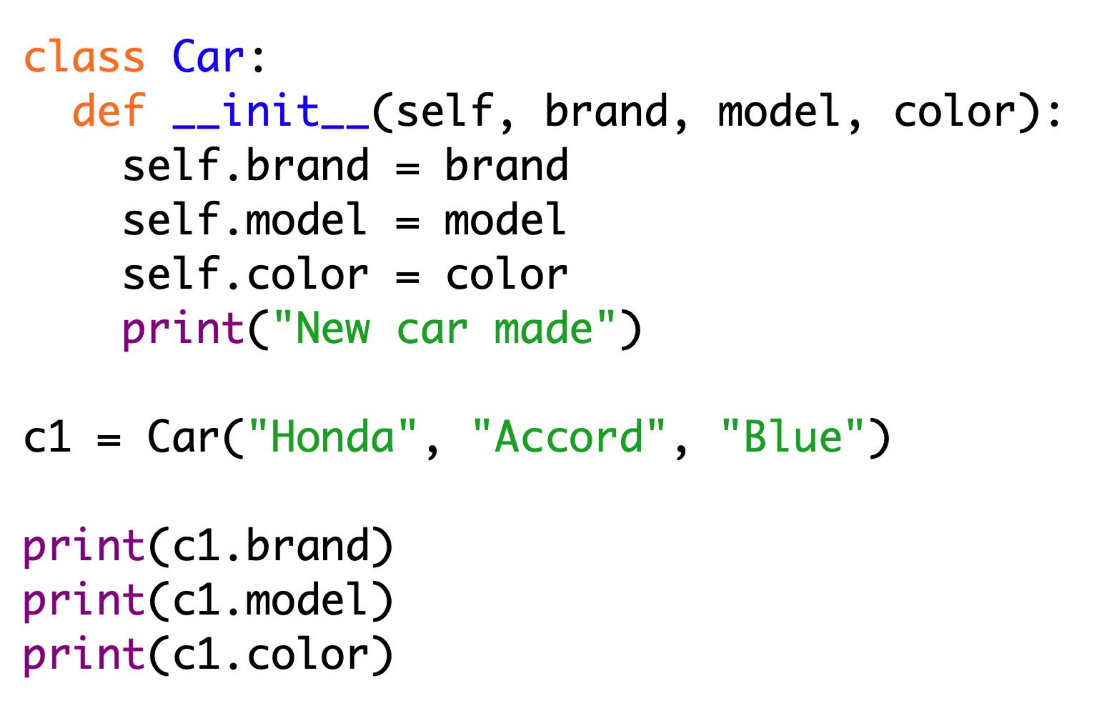
Constructors



```
def __init__(self):
    print("New object being made!")
```

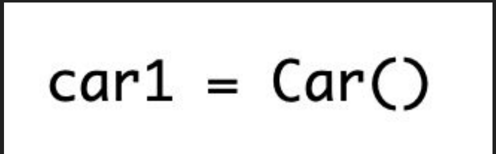
- All classes have an `__init__()` function that is executed one time when an object is created
- It is used to assign values to object properties
- This function requires single argument `self` which is a reference to the instance that is being created.
- While you technically can use a different name instead of self, it's strongly recommended to stick to the convention and use self.

Defining a class



```python
class Car:
  def __init__(self, brand, model, color):
    self.brand = brand
    self.model = model
    self.color = color
    print("New car made")

c1 = Car("Honda", "Accord", "Blue")

print(c1.brand)
print(c1.model)
print(c1.color)
```

Creating objects from a class

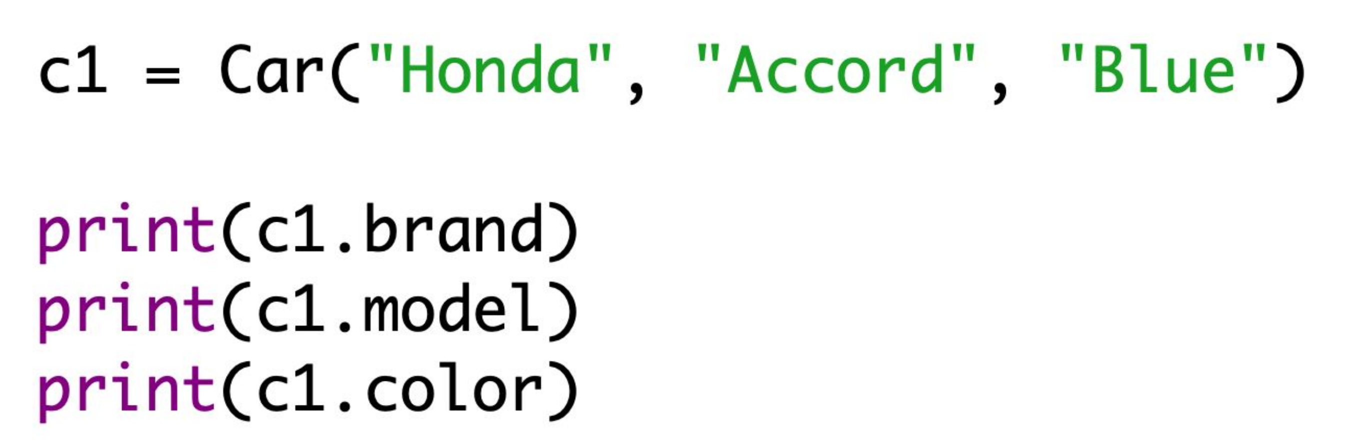


```
car1 = Car()
```

- Creating an object from a class is called **instantiation.**
- We create an object by using the name of the class followed by parenthesis.
- The variable `car1` is holding the memory address of where the object will be stored.
- You must define your class before you try to create an object!

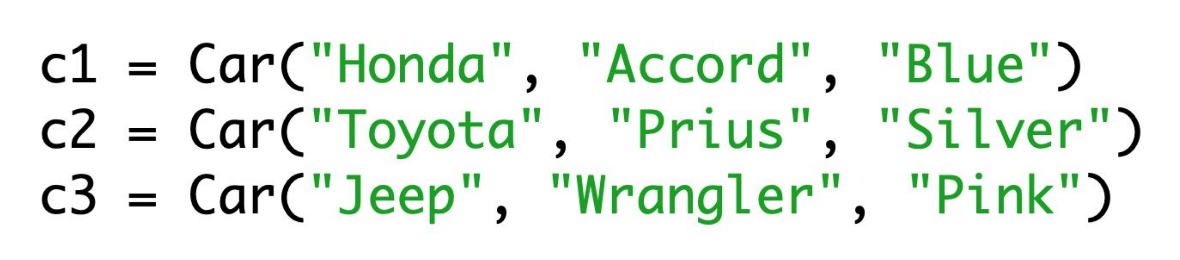
Accessing attributes within a class

— To access data within a class, we use the “dot syntax”



```
c1 = Car("Honda", "Accord", "Blue")

print(c1.brand)
print(c1.model)
print(c1.color)
```

Creating Multiple Instances

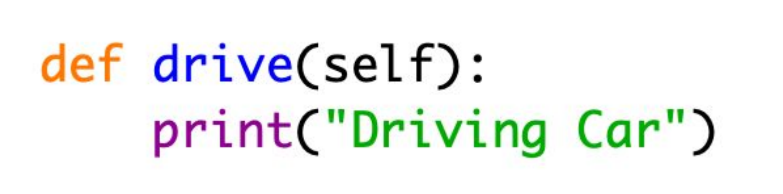


```
c1 = Car("Honda", "Accord", "Blue")
c2 = Car("Toyota", "Prius", "Silver")
c3 = Car("Jeep", "Wrangler", "Pink")
```

- One of the biggest advantages of defining classes is that you can make as many objects as you would like!
- Each instance of a class has its own set of data attributes
  - Classes allow you to make many different independent copies

Methods within Classes

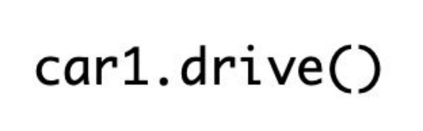
- In addition to attaching values to an object we can also attach functions to our objects as well.



```python
def drive(self):
    print("Driving Car")
```

- The function is designed to accept the 'self' argument, just like the constructor function does. We call functions defined in this way as 'methods' of the object

- To use the method, we can use the dot notation to write



```python
car1.drive()
```

Pssst… we’ve actually been using classes all semester!

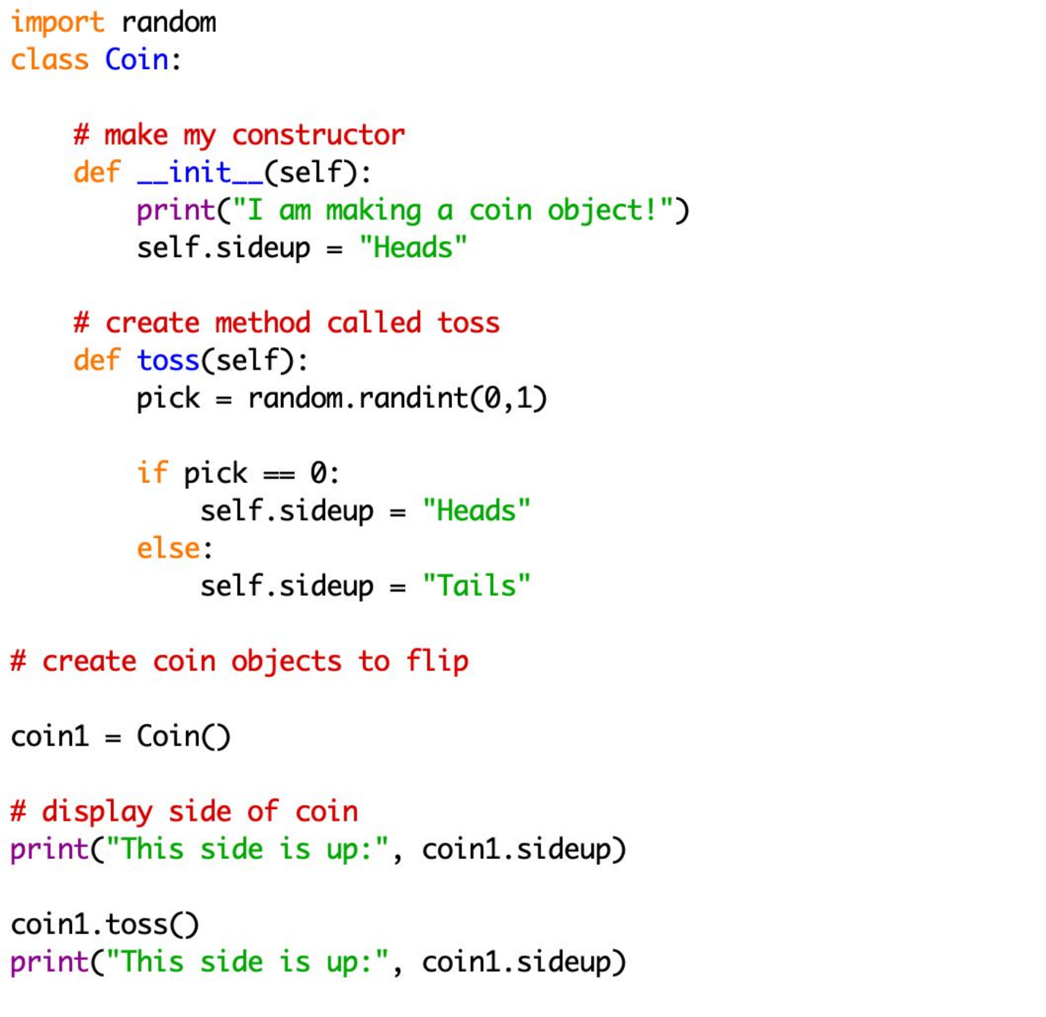
- Floats
- Strings
- Lists
- Dictionaries
- Booleans

And all these classes have methods (like functions) that we call using dot syntax.

```
list.append()
str.split()
dict.keys()
```

Programming Challenge

- Design a class called **Coin** that simulates a coin being flipped.
- The class should have an attribute called “sideup” to store whether the coin is “Heads” or “Tails”
- The class should have a method to toss the coin and randomly choose between heads or tails.

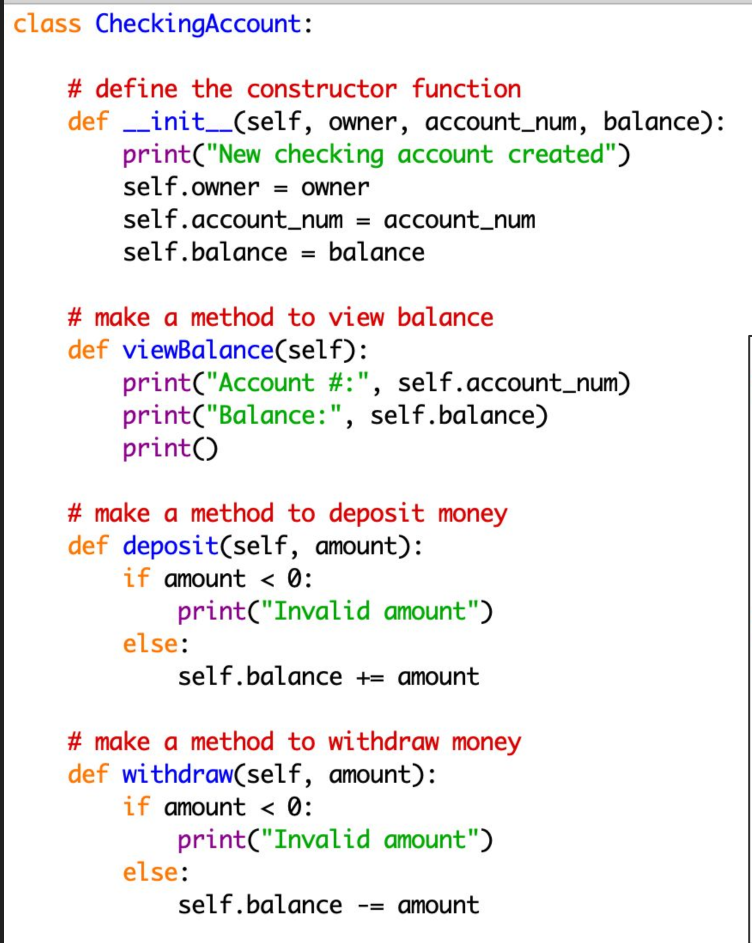


```
import random
class Coin:

    # make my constructor
    def __init__(self):
        print("I am making a coin object!")
        self.sideup = "Heads"

    # create method called toss
    def toss(self):
        pick = random.randint(0,1)

        if pick == 0:
            self.sideup = "Heads"
        else:
            self.sideup = "Tails"

# create coin objects to flip

coin1 = Coin()

# display side of coin
print("This side is up:", coin1.sideup)

coin1.toss()
print("This side is up:", coin1.sideup)
```

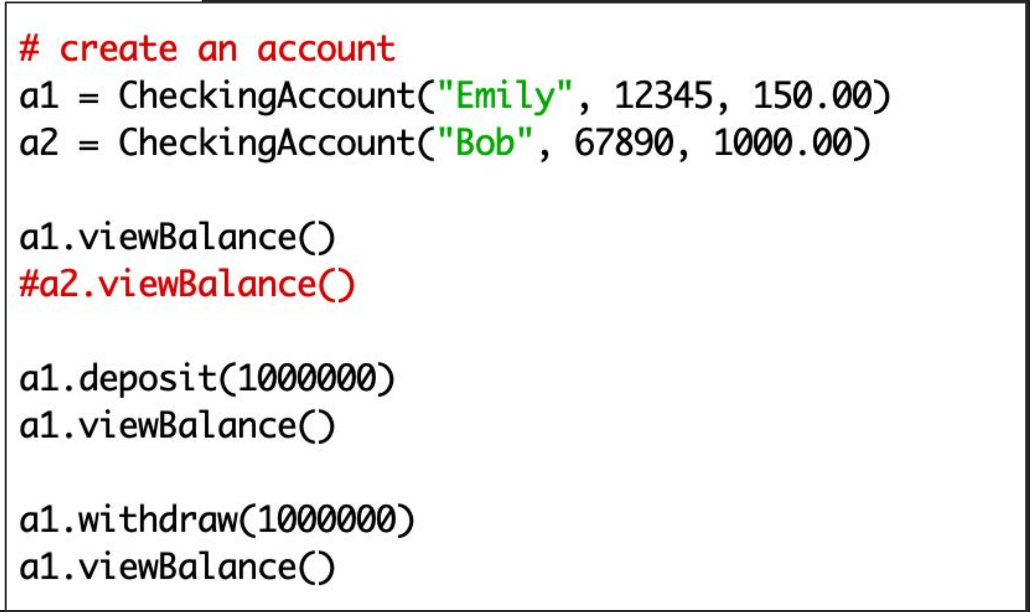
Programming Challenge

Design a class called **CheckingAcccount** which has the following:

- A constructor that accepts 4 arguments: an owner, account number, and balance
- A method called ”view_balance" — this method should accept no arguments and prints the account number and balance
- A method called “withdraw” with 1 argument that removes a specified amount of money from the account
- A method called “deposit” with 1 argument that adds a specified amount of money to the account



```
class CheckingAccount:

    # define the constructor function
    def __init__(self, owner, account_num, balance):
        print("New checking account created")
        self.owner = owner
        self.account_num = account_num
        self.balance = balance

    # make a method to view balance
    def viewBalance(self):
        print("Account #:", self.account_num)
        print("Balance:", self.balance)
        print()

    # make a method to deposit money
    def deposit(self, amount):
        if amount < 0:
            print("Invalid amount")
        else:
            self.balance += amount

    # make a method to withdraw money
    def withdraw(self, amount):
        if amount < 0:
            print("Invalid amount")
        else:
            self.balance -= amount
```



```
# create an account
a1 = CheckingAccount("Emily", 12345, 150.00)
a2 = CheckingAccount("Bob", 67890, 1000.00)

a1.viewBalance()
#a2.viewBalance()

a1.deposit(1000000)
a1.viewBalance()

a1.withdraw(1000000)
a1.viewBalance()
```