# Python Track | Codecademy

## Modifying a List

**n.append(item)** adds item at end of list

**n.pop(index)** removes item at index and returns it to you

**n.remove(item)** removes item if found

**del(n[index])** will remove item at given index, but won't return it

**" ".join(n)** combines items in a list into a string, and whatever precedes the period is inserted between the items e.g. spaces in this case

## Iterating through List

**for item in n:** is simple but can cause problems when modifying items

**for item in range()** function defaults are 0 start and 1 step, and stop means up to but not including stop

* **range(stop)** e.g. range(len(n))
* **range(start, stop)**
* **range(start, stop, step)** e.g. range(0, len(n), -1) returns list backwards

Note to self: Do notice whether you are referring to list or list[item]

## Nesting Loops

Example: you have a list containing two lists

n = [[1, 2, 3], [4, 5, 6, 7, 8, 9]]  
  
def flatten(lists):  
    results = []  
    for lst in lists:  
        for numbers in lst:  
    results.append(numbers)  
    return results  
  
print flatten(n)  
  
==> [1, 2, 3, 4, 5, 6, 7, 8, 9]

## Battleship

**randint(low, high)** function from the random module can be imported to generate random numbers given an inclusive range.

from random import randint  
   
board = []  
   
for x in range(5):  
    board.append(["O"] \* 5)  
   
def print\_board(board):  
    for row in board:  
        print " ".join(row)  
   
print "Let's play Battleship!"  
print\_board(board)  
   
def random\_row(board):  
    return randint(0, len(board) - 1)  
   
def random\_col(board):  
    return randint(0, len(board[0]) - 1)  
   
ship\_row = random\_row(board)  
ship\_col = random\_col(board)  
# Only for debugging purposes:  
# print ship\_row  
# print ship\_col  
   
# Everything from here on should go in your for loop!  
# Be sure to indent four spaces!  
for turn in range(4):  
   
    guess\_row = int(raw\_input("Guess Row:"))  
    guess\_col = int(raw\_input("Guess Col:"))  
    # If player guesses the right location  
    if guess\_row == ship\_row and guess\_col == ship\_col:  
        print "Congratulations! You sunk my battleship!"  
        break  
    else:  
        # If player is wrong on last turn  
        if turn == 3:  
            print "Game Over"  
        # If player guesses out of range  
        elif (guess\_row < 0 or guess\_row > 4) or (guess\_col < 0 or guess\_col > 4):  
            print "Oops, that's not even in the ocean."  
        # If player guesses that already  
        elif(board[guess\_row][guess\_col] == "X"):  
            print "You guessed that one already."  
        else:  
            print "You missed my battleship!"  
            board[guess\_row][guess\_col] = "X" # Be careful about indentation here, kept running into IndexError: list index out of range because I was missing an indent  
   
    print "Turn ", turn + 1  
    print\_board(board)

## Loops

**while** loop continues to execute while a condition is true instead of if it is true  
be careful - make sure the while condition can be reached, otherwise your loop could go on forever and become an infinite loop i.e. a loop which never exits, which could crash your computer / browser!  
**break** means "exit the current loop," prior to it you can write define the stopping condition using an if statement  
**while/else** the else block will execute when the loop condition is evaluated to False, unless the loop exits as a result of a break  
for i in range(10): This kind of loop is useful when you want to do something a certain number of times (not inclusive thoguh, this example goes from 0-9)

**for c in thing:** Using a for loop, you can loop through each individual character in a string, which is useful for string manipulation  
**,** keeps print statement in the same line, similar to concatenation using + except it adds a space  
**for animals in zoo:** Perhaps the most useful (and most common) use of for loops is to go through a list.  
**for key in d:** Using a for loop, you can also loop through a dictionary to get the keys and or the values  
 **print key, d[key]** In this example, key is the key and d[key] is the associated value

**enumerate(sequence, start=0)** Built-in enumerate function supplies a corresponding index to each list element  
**for index, season in enumerate(seasons):** During the loop, index increases as item moves to next in sequence  
 **print index+1, season** Index usually starts at 0.   
**list(enumerate(seasons, start=1))** Output: [(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]

Equivalent to:

def enumerate(sequence, start=0):  
    n = start for elem in sequence:  
        yield n, elem  
        n += 1

**zip** will create pairs of elements when passed two (or more) lists and will stop at the end of the shorter list  
**print zip(list\_a, list\_b)** Output: [(3, 2), (9, 4), (17, 8), (15, 10), (19, 30)]  
**for a, b in zip(list\_a, list\_b):** iterates over two lists at once  
**for/else** the else statement is executed after the for, but only if the for ends normally—that is, not with a break

Example:

for f in fruits:  
    if f == 'tomato':  
        print 'A tomato is not a fruit!' # (It actually is.)  
        # break here would make the else not run  
    print 'A', f  
    else:  
        print 'A fine selection of fruits!'

## Classes

Python is an **object-oriented programming language**, which means it manipulates programming **constructs** called **objects**. You can think of an object as a single data structure that contains data as well as functions; functions of objects are called **methods**. For example, any time you **call**

len("snake")

Python is checking to see whether the string object you passed it has a length, and if it does, it returns the **value** associated with that **attribute**. When you call

my\_dict.items()

Python checks to see if my\_dict has anitems() method (which all dictionaries have) and executes that method if it finds it.

But what makes "snake" a string and my\_dict a dictionary? The fact that they're **instances** of the str and dict classes, respectively. A **class** is just a way of organizing and producing objects with similar attributes and methods.

class Fruit(object):  
    """A class that makes various tasty fruits."""  
    def \_\_init\_\_(self, name, color, flavor, poisonous):  
        self.name = name  
        self.color = color  
        self.flavor = flavor  
        self.poisonous = poisonous  
  
def description(self):  
    print "I'm a %s %s and I taste %s." % (self.color, self.name, self.flavor)  
  
def is\_edible(self):  
    if not self.poisonous:  
        print "Yep! I'm edible."  
    else:  
        print "Don't eat me! I am super poisonous."  
  
lemon = Fruit("lemon", "yellow", "sour", False)  
  
lemon.description()  
lemon.is\_edible()

A basic class consists only of the class **keyword**, the **name** of the class, and the **class  from which the new class inherits** in parentheses. By convention, user-defined Python class names start with a capital letter.

Class syntax:

class ClassName(object):  
    def \_\_init\_\_(args):  
        # Set self.args = args

**object** in parentheses is what we use because we want our classes to inherit the object class.  This means that our class has all the properties of an object, which is the simplest, most basic class. Later we'll see that classes can inherit other, more complicated classes.

**def \_\_init\_\_():** function is required for classes, and it's used to **initialize** the objects it creates. It always takes at least one argument, self. Think of it as the method that "boots up" a class' instance object: the init bit is short for "initialize."

**Self** is, by convention, the first parameter passed to \_\_init\_\_(). Python will use the first argument that \_\_init\_\_() gets to refer to the instance object being created; this is why it's often called self, since this parameter gives the object being created its identity.

If you add **additional arguments**—for instance, a **name** and **age** for your animal—setting each of those equal to **self.name** and **self.age** in the body of \_\_init\_\_()will make it so that when you create an instance object of your Animal class, you need to give each instance a name and an age (unless you pre-define them in Animal class), and those will be associated with the particular **instance** you create.

We can access attributes of our objects using **dot notation.** Below we create a class Animal with an attribute name. Outside the class definition, we create a new instance of Animal named zebra and access that attribute using zebra.name.

class Animal(object):  
    def \_\_init\_\_ (self, name):  
        self.name = name  
  
zebra = Animal("Robert")  
print zebra.name

Another important aspect of Python classes is **scope**. The scope of a variable is the context in which it's visible to the program. Not all variables are accessible to all parts of a Python program at all times. When dealing with classes, you can have variables that are available everywhere (**global variables**), variables that are only available to members of a certain class (**member variables**), and variables that are only available to particular instances of a class (**instance variables**).The same goes for functions: some are available everywhere, some are only available to members of a certain class, and still others are only available to particular instance objects.

When a class has its own functions, those functions are called **methods**. You've already seen one such method:\_\_init\_\_(). But you can also define your own methods!

class Animal(object):  
    """Makes cute animals."""  
    is\_alive = True  
    def \_\_init\_\_(self, name, age):  
        self.name = name  
        self.age = age  
      
    def description(self):  
        print self.name  
        print self.age  
  
hippo = Animal('potamus', '17')  
hippo.description()

A class can have any number of **member variables**. These are variables that are available to all members of a class and appear before the \_\_init\_\_() method. You can access these member variables using dot notation. You can even change the default value stored in a member variable for a particular instance of the class, for example hippo.is\_alive = False will change the value for hippo but not for other instances of the class.

More realistic example:

class ShoppingCart(object):  
    """Creates shopping cart objects  
    for users of our fine website."""  
    items\_in\_cart = {}  
    def \_\_init\_\_(self, customer\_name):  
        self.customer\_name = customer\_name  
  
    def add\_item(self, product, price):  
        """Add product to the cart."""  
        if not product in self.items\_in\_cart:  
            self.items\_in\_cart[product] = price  
            print product + " added."  
        else:  
            print product + " is already in the cart."  
  
    def remove\_item(self, product):  
        """Remove product from the cart."""  
        if product in self.items\_in\_cart:  
            del self.items\_in\_cart[product]  
            print product + " removed."  
        else:  
            print product + " is not in the cart."  
              
my\_cart = ShoppingCart('Nina')  
my\_cart.add\_item('scissors', 2.50)

## Inheritance

**Inheritance** is the process by which one class takes on the attributes and methods of another, and it's used to express an **is-a**relationship. For example, a Panda is a bear, so a Panda class could inherit from a Bear class. Panda can have access to the methods in Bear class via inheritance, even if you don't define a certain method in Panda.

Example:

class Customer(object):  
    """Produces objects that represent customers."""  
    def \_\_init\_\_(self, customer\_id):  
        self.customer\_id = customer\_id  
  
    def display\_cart(self):  
        print "I'm a string that stands in for the contents of your shopping cart!"  
  
class ReturningCustomer(Customer):  
    """For customers of the repeat variety."""  
    def display\_order\_history(self):  
        print "I'm a string that stands in for your order history!"  
  
monty\_python = ReturningCustomer("ID: 12345")  
monty\_python.display\_cart()  
monty\_python.display\_order\_history()

**Inheritance** mechanism implements a **type and subtype** relationship between classes.

In Python, inheritance looks like this:

*class Subclass(Superclass):*  
*pass*

where **Subclass** (or derived class or child class) is the new class you're making and **Superclass** (or base class or parent class) is the class from which that new class inherits.

**pass** keyword doesn't do anything, but it's useful as a placeholder in areas of your code where Python expects an expression.

Subclasses can inherit from another to not only take on the methods and attributes of its parent, but to **override** one or more of them.

class Employee(object):  
    def \_\_init\_\_(self, name):  
        self.name = name  
    def greet(self, other):  
        print "Hello, %s" % other  
  
class CEO(Employee):  
    def greet(self, other):  
        print "Get back to work, %s!" % other.name  
  
ceo = CEO("Emily")  
emp = Employee("Steve")  
emp.greet(ceo)  
# Hello, Emily  
ceo.greet(emp)  
# Get back to work, Steve!

**super()** call allows you to directly access the attributes or methods of a superclass from a subclass. This is useful in cases where you're working with a subclass and realize you've overwritten a method or attribute defined in the superclass that you actually need.

class Subclass(Superclass):  
   def m(self):  
       return super(Subclass, self).m() # do not include self, but include other args

Another example:

class Employee(object):  
    """Models real-life employees!"""  
    def \_\_init\_\_(self, employee\_name):  
        self.employee\_name = employee\_name   
  
    def calculate\_wage(self, hours):  
        self.hours = hours  
        return hours \* 20.00  
  
class PartTimeEmployee(Employee):  
    def calculate\_wage(self, hours):  
        self.hours = hours  
        return hours \* 12.00  
    def full\_time\_wage(self, hours):  
        return super(PartTimeEmployee, self).calculate\_wage(hours) # note the syntax  
  
milton = PartTimeEmployee('milton')  
print milton.employee\_name   
print milton.calculate\_wage(10)  
print milton.full\_time\_wage(10)

Instantiating an object:

class Triangle(object):  
    number\_of\_sides = 3  
    def \_\_init\_\_(self, angle1, angle2, angle3):  
        self.angle1 = angle1 # always start with self dot notation  
        self.angle2 = angle2  
        self.angle3 = angle3  
      
    def check\_angles(self): # no need to include extra parameters unless asker for new parameters  
        if self.angle1 + self.angle2 + self.angle3 == 180:  
            return True  
        else:  
            return False  
              
class Equilateral(Triangle):  
    angle = 60  
    def \_\_init\_\_(self):  
        self.angle1 = self.angle  
        self.angle2 = self.angle  
        self.angle3 = self.angle  
          
my\_triangle = Triangle(90, 30, 60)  
print my\_triangle.number\_of\_sides # don't need self dot notation outside of class object  
print my\_triangle.check\_angles() # don't forget () notation for methods  
  
my\_equilateral = Equilateral()  
print my\_equilateral.angle1 # can refer to any of the member or instance variables  
print my\_equilateral.check\_angles()

## Class Review

Classes can be very useful for storing complicated objects with their own methods and variables.

Defining a **class**:

class ClassName(object):  
    *# class statements go here*

Creating a new **instance** of a class (outside of the class definition):

newObject = ClassName()

Creating **member variables** (they store information about and belong to each class object) and assigning them initial values:

class ClassName(object):  
    variable = "initialValue"

Defining **\_\_init\_\_()**must always start with keyword self - this is how the object keeps track of itself internally - but we can pass additional member variables after that by using dot notation:

def \_\_init\_\_(self, new\_variable):  
    self.new\_variable = new\_variable

Calling or accessing variables works the same (using **dot notation**) whether the class member variables are created within the class or are passed into the new object at initialization:

print newObject.variable  
print newObject.new\_variable

Creating class **methods**is identical to defining any other function, except that it is written inside of a class definition (like \_\_init\_\_ function, you need to provide self as the first argument of any class method):

class Car(object):  
    condition = "new"  
    def \_\_init\_\_(self, model, color, mpg):  
        self.model = model  
        self.color = color  
        self.mpg   = mpg  
    def display\_car(self):  
        print "This is a %s %s with %s MPG." % (self.color, self.model, self.mpg)    
  
my\_car = Car("DeLorean", "silver", 88)

Calling class methods is like calling member variables, but with using **parentheses** behind the dot notation:

print my\_car.display\_car()

Modifying member variables is similar to how to we initialize them - useful when we want to change the value a variable takes on based on something that happens inside of a class method.

    # continue after display\_car method  
    def drive\_car(self):  
        self.condition = 'used'  
  
my\_car = Car("DeLorean", "silver", 88)  
print my\_car.condition  
my\_car.drive\_car()  
print my\_car.condition

Using **inheritance** to create more complicated classes that inherit variables or methods from their parent classes while being able to include additional variables or methods:

class Subclass(Superclass):  
    # new variables and functions go here

Normally we use object as the parent class because it is the most basic type of class, but by specifying a different class, we can inherit more complicated functionality.

**Redefining or overriding** a subclass method is as simple as including a definition for that function inside the subclass; this version will take precedence over the inherited version:

# continue after Car class  
class ElectricCar(Car):  
    def \_\_init\_\_(self, model, color, mpg, battery\_type):  
    # have to redefine all attributes if new \_\_init\_\_() method      
        self.model = model  
        self.color = color  
        self.mpg   = mpg  
        self.battery\_type = battery\_type  
  
my\_car = ElectricCar("DeLorean", "silver", 88, "molten salt")  
print my\_car.battery\_type

Overriding built-in **\_\_repr\_\_()** method, which is short for representation; by providing a return value in this method, we can tell Python how to represent an object of our class:

class Point3D(object):  
    def \_\_init\_\_(self, x, y, z):  
        self.x = x  
        self.y = y  
        self.z = z  
    def \_\_repr\_\_(self):  
        return "(%d, %d, %d)" % (self.x, self.y, self.z)  
  
my\_point = Point3D(1, 2, 3)  
print my\_point

For more information on \_\_repr\_\_ and other special methods see this [Python documentation](http://docs.python.org/2/reference/datamodel.html#object.__repr__). Note the slight difference between the \_\_repr\_\_ and \_\_str\_\_methods.