Python Workshop 2 - All about OO

What does OO stand for?

What are the two main things that all objects have?

What is a class?

Write the basic structure to define a class below:

What does the **__init__** function do?

What does the **__str__** function do?

Why do these two functions have double underscores in their names?

What functions would I need to define if I wanted to use the comparison operators (<, >, ==, etc.) on a custom class?

If I had a class, Apple, how would I make an instance of that class? (This class takes no arguments to its initializer.)

If I needed to use the function **sqrt** from the library **Math**, what two steps would I need to take to

use it?

Some random functions

http://docs.python.org/library/random.html

random.**randint**(*a*, *b*)

Return a random integer *N* such that `a <= N <= b`.

random.**choice**(*seq*)

Return a random element from the non-empty sequence *seq*. If *seq* is empty, raises **IndexError**.

random.**random**()

Return the next random floating point number in the range [0.0, 1.0).

random.**uniform**(*a*, *b*)

Return a random floating point number *N* such that `a <= N <= b` for `a <= b` and `b <= N <= a` for `b < a`.

Exercise

You will be creating a class called Individual, which models an individual haploid organism. I will detail below the different function associated with this class, I recommend you implement them in the order I talk about them, however I would look through them all before you start to get an idea of where we are going.

As you complete each exercise make some code that tests each function for this class. Make sure to comment heavily, it will help you remember what is going on later.

1. Initializing the class

You should make the class take no options, and generate a random genome of ten nucleotides (A, T, C, G).

```
>>> myInd = Individual()
>>> print(myInd.genome)
['A', 'G', 'G', 'T', 'T', 'C', 'A', 'T', 'T', 'A']
```

Once you have done this, make the initializer for the class ***optionally*** take a genome that has already been made, and use that instead of generating one. (Make sure that the code above still works after you've done this, remember this is an optional argument.)

```
>>> myGenome = ['T', 'A', 'T', 'G', 'C', 'T']
>>> myInd = Individual(myGenome)
>>> print(myInd.genome)
['T', 'A', 'T', 'G', 'C', 'T']
```

2. Making printing easy

Implement the __str__ function for the Individuals class to make printing easy. Make it return a string that contains the genome of the Individual.

```
>>> myInd = Individual()
>>> print(myInd)
Individual(['A', 'G', 'G', 'T', 'T', 'C', 'A', 'T', 'T', 'A'])
```

3. Calculating fitness

If we want to ask any interesting questions about this population then we need to be able to determine the fitness of each individual. We could do this any number of ways. For now lets calculate fitness as the GC content of the genome. So, a genome consisting entirely of CGs will have a fitness of 1, while a genome consisting of entirely ATs will have a fitness of 0. Make a function that calculates fitness of the individual and returns it.

```
>>> myInd = Individual()
>>> print(myInd.genome)
['A', 'C', 'G', 'C', 'G', 'G', 'C', 'T', 'G', 'C']
```

```
>>> myFitness = myInd.fitness()
>>> print(myFitness)
0.8
```

Once you have calculated the fitness store it in the individual, so you don't need to ever calculate fitness for a single individual twice. Modify the __str__ function to also print out the Individual's fitness.

```
>>> myInd = Individual()
>>> print(myInd)
Individual(genome =
['A', 'G', 'G', 'T', 'T', 'C', 'A', 'T', 'T', 'A'], fitness =
0.3)
```

4. Mutations

Make a function that mutates an Individual's genome. Each site should have a probability of being mutated equal to 0.1. If a site does get mutated it has an equal chance of changing to any one of the four nucleotides (this means that a mutation can have no effect, if we mutate from a T to a T for example).

```
>>> myInd = Individual()
>>> print(myInd)
Individual(genome =
['A', 'C', 'G', 'C', 'G', 'G', 'C', 'T', 'G', 'C'], fitness =
0.8)
>>> myInd.mutate()
>>> print(myInd)
Individual(genome =
['A', 'T', 'T', 'C', 'T', 'C', 'C', 'G', 'C', 'C'], fitness =
0.6)
```

5. Crossing over

Next we want two Individuals to be able to produce an offspring. Make a function that takes an Individual as the other parent, and returns a new Individual offspring. Cross over the two genomes at the half way point and randomly pick one of the resulting genomes to be that of the offspring.

```
>>> parent1 = Individual()
>>> print(parent1)
Individual(genome =
['C', 'G', 'A', 'G', 'T', 'C', 'G', 'A', 'T', 'C'], fitness =
0.6)
>>> parent2 = Individual()
>>> print(parent2)
Individual(genome =
['T', 'T', 'A', 'G', 'A', 'T', 'A', 'A', 'C', 'C'], fitness =
0.3)
>>> offspring = parent1.crossover(parent2)
```

```
>>> print(offspring)
Individual(genome =
['C', 'G', 'A', 'G', 'T', 'T', 'A', 'A', 'C', 'C'], fitness =
0.5)
>>> offspring2 = parent1.crossover(parent2)
>>> print(offspring2)
Individual(genome =
['T', 'T', 'A', 'G', 'A', 'C', 'G', 'A', 'T', 'C'], fitness =
0.4)
```

Look at this code for a while before you start writing the function. Draw the genomes out and make a cartoon of what is happening biologically, don't think about the code. Once you understand that then start implementing it.

Once you have that working modify the crossover function such that it crosses over in a random place, not always exactly in the middle.

```
>>> offspring = parent1.crossover(parent2)
>>> print(offspring)
Individual(genome =
['C', 'G', 'A', 'G', 'A', 'T', 'A', 'A', 'C', 'C'], fitness =
0.5)
>>> offspring2 = parent1.crossover(parent2)
>>> print(offspring2)
Individual(genome =
['C', 'G', 'A', 'G', 'T', 'C', 'G', 'A', 'C', 'C'], fitness =
0.7)
```

6. Making it run

Now you have a complete Individual! Lets simulate a population of 2 Individuals going for some time, this should all be done in a separate file from the one you have the Individual class written in. Start by making 2 Individuals, with random genomes. Then make a loop that crosses those 2 Individuals twice (yielding two offspring), and mutates each offspring. Then use these offspring as the parents in the next generation (here a generation is just an iteration of the loop). Each generation print out the mean fitness of the population. Make this go for 50 generations.

Plot the mean fitness each generation in excel. Does the population evolve with some direction? What if you do this with 3 Individuals, and only the 2 with the highest fitness get to mate each generation?

7. Making it more complex

The following are just some minor tweaks you could perform to make your Individual more generalized. Don't spend too much time on these, they are just for some breadth.

1. Sample the number of mutations an individual will have from a Poisson distribution (see the python random.poisson docs), with lambda = 2.

2. Make the mean number of mutations an **optional** input parameter to the mutate() function.
3. Make the number of crossover events be sampled from a Poisson distribution, with the mean an input parameter.
4. Make the length of an Individual's genome an **optional** input parameter, make 10 the default.
5. Make an alternate fitness function that calculates fitness based on something other than GC content. Pick anything. Really. Anything. You're thinking too hard. Just do it.
6. Make your program from step 6 more formal. Make it easy to vary the number of individuals in the population and the number of generations.

At home exercise

The homework exercise is simpler than the in class one, so you could give this a try if you get stuck on Individual. For this project you are going to be making a linked list. In reality you will be making a class, called LinkedNode, that has some value, and points to the next node in a list.

There is a python script called “linkedTest.py”, this file will test each of these functions for you. Once you have finished (or even as you go) import your file at the top of linkedTest.py and give it a run. Note, this will only work if you have named the functions and class exactly the way I specify below, including capitalization.

1. Making a node

Make a class, LinkedNode, that takes one input parameter to get initialized, and stores that value.

```
>>> node1 = LinkedNode(10)
>>> print(node1.value)
10
>>> node2 = LinkedNode(“apples”)
>>> print(node2.value)
apples
```

2. Making a node point to another node

Make the node optionally take a second argument that is a node that should point to this node.

```
>>> node1 = LinkedNode(10)
>>> node2 = LinkedNode(“apples”, node1)
>>> print(node1.next)
<LinkedNode instance at 0x022CA9E0>
>>> print(node2.next)
None
>>> print(node1.next.value)
apples
```

3. Changing links

Make a function **addNode** that takes a node as an argument, this function should change the node that *self* points to to the input argument.

```
>>> node1 = LinkedNode(10)
>>> node2 = LinkedNode(“apples”, node1)
>>> print(node1.next.value)
apples
```

```
>>> node3 = LinkedNode(17)
>>> node2.addNode(node3)
>>> print(node2.next.value)
17
>>> node1.addNode(node3)
>>> print(node1.next.value)
17
```

3. Printing the whole list

Define the **__str__** function, such that it prints out a single string that lists the values for all nodes in the list, separated by commas.

```
>>> node1 = LinedNode(10)
>>> newNode = LinkedNode(11, node1)
>>> newNode = LinkedNode(12, newNode)
>>> newNode = LinkedNode(13, newNode)
>>> newNode = LinkedNode(15, newNode)
>>> newNode = LinkedNode(14, newNode)
>>> print(node1)
10, 11, 12, 13, 15
```

4. Getting the *ith* node

Make a function, **index**, that takes a number, n, and returns the value of the nth node after *self*. If there are less than n nodes, then it should return None.

```
>>> print(node1.index(1))
11
>>> print(node1.index(3))
13
>>> print(node1.index(2))
12
>>> print(node1.index(15))
None
```

5. Removing a node

Make a function, **remove**, that takes an index of a node to remove from the list, and removes it. Don't worry about ever removing the first node. If the index is longer than the list then the list should remain unchanged.

```
>>> print(node1)
10, 11, 12, 13, 15, 14
>>> node1.remove(1)
>>> print(node1)
10, 12, 13, 15, 14
>>> node1.remove(4)
>>> print(node1)
```

```
10, 12, 13, 15
>>> node1.remove(10)
>>> print(node1)
10, 12, 13, 15
```

6. Appending to the list

Make a function, **append**, that takes a value and adds it to the end of the list.

```
>>> print(node1)
10, 11, 12, 13, 15, 14
>>> node1.append(72)
>>> print(node1)
10, 11, 12, 13, 15, 14, 72
```

7. inserting a node

Make a function, **insert**, that takes an index and a value, and adds a new node with that value at the specified index in your list. If the provided index is longer than the list do not add it. Don't worry about ever inserting at the first node.

```
>>> print(node1)
10, 11, 12, 13, 15, 14
>>> node1.insert(72, 2)
>>> print(node1)
10, 11, 72, 12, 13, 15, 14
>>> node1.insert(5, 5)
>>> print(node1)
10, 11, 72, 12, 13, 5, 15, 14
>>> node1.insert(65, 10)
>>> print(node1)
10, 11, 72, 12, 13, 5, 15, 14
```

8. Re-trying

Implementing this code is much easier if you use recursion. A recursive function is simply a function that calls itself. you should try to implement 3-7 using recursion instead of loops. The answers include both non-recursive, and recursive solutions.

To practice recursion to to implement Factorial and Fibonacci from the workshop 1 homework using recursion.

This section is not critical to classes or objects, and is more about learning a new coding technique. Don't spend too much time trying to make this work.