CS Prof. writes Python code that produces Nissan Leafs. (“Leaves?” p. 42)

CS 5 Green Today

Students Object to Classes. "They serve no function and we disagree with the methods," say students.

(Clariment, AP): Students in CS 5 say that they object to classes. "We're overloaded!" said one student, "and we want to underscore underscore our concerns." Another student spokesperson said "The professors are def __init__ely hoping this is something that will just float away, but they can't string us along forever. We have a long list of issues and if the profs don't understand them, they should look them up in a dictionary," said a student. "We sure wish the students were mutable!" said one professor. Students and professors eventually agreed on a tuple of ways to __repr__ their relationship.
Learning Goals
• Explain Markov models for simulation
• Practice classes
An Ant Class

```python
from Vector import *

class Ant:
    def __init__(self, pos):
        self.position = pos

    def moveTowards(self, other):

>>> abes_position = Vector(0, 0)
>>> abe = Ant(abes_position)
>>> bess = Ant(Vector(100, 0))
>>> abe.moveTowards(bess)
(100, 0)!? That’s practically in the Dutch Ant-illes!
```

Hey, draw me a picture of Abe and Bess!

I’m feeling strong ant-ipathy for ant puns!
abe = Ant(Vector(0, 0))
bess = Ant(Vector(0, 100))
cziggy = Ant(Vector(100, 100))
dizzy = Ant(Vector(100, 0))

while True:
    abe.moveTowards(bess)
bess.moveTowards(cziggy)
cziggy.moveTowards(dizzy)
dizzy.moveTowards(abe)

Ugh! What if there were 1000 ants, or even some variable n number of ants!
The Age of Abstraction

Abstraction in CS

```python
class list:
    def __init__(self):
    def append(self, item):
    def __repr__(self):
    def __getitem__(self, index):
    def __setitem__(self, index, value):
```

```python
>>> x = []
>>> x.append(42)
>>> x
[42]
>>> x[0]
42
>>> x[0] = 67

>>> x = list()
>>> x.append(42)
>>> x.__repr__()
[42]
>>> x.__getitem__(0)
42
>>> x.__setitem__(0, 67)
```
HW: Markov text generation

1st order

Training text
I like cookies. I like spam. I am happy. Spam is good.

Learning phase
Starters: [("I",), ("I",), ("I",), ("Spam",)]
Dictionary:

```python
{
    ("I",): ["like", "like", "am"],
    ("like",): ["cookies.", "spam."],
    ("cookies.",): ["I"],
    ("spam.",): ["I"],
    ("am",): ["happy."],
    ("happy.",): ["Spam"],
    ...
}
```
HW: Markov text generation

2\textsuperscript{nd} order

Training text
I like cookies. I like spam. I am happy. Spam is very good.

Learning phase
Starters:
[ ("I", "like"), ("I", "like"),
  ("I", "am"), ("Spam", "is")
]

Dictionary:
{
  ("I", "like"): ["cookies.", "spam."],
  ("I", "am"): ["happy."],
  ("is", "very"): ["good."],
  ...
}
Markov models in biology

- Gene finding
- Nucleotide substitution models
- Sequence similarity search
- Modeling animal behavior
Markov models in bio 52...

Cholera comparative genomics

AAAATA: 0.021 AAAAAC: 0.020 AAAAAG: 0.013 AAAAAT: 0.019...
AACAAC: 0.020 AACAAG: 0.023 AACAAT: 0.031...
AAGAAA: 0.057 AAGAAC: 0.017 AAGAAG: 0.033 AAGAAT: 0.020...
AATAAA: 0.049 AATAAC: 0.016 AATAAG: 0.016 AATAAT: 0.034...
ACAAAA: 0.022 ACAAAAC: 0.015 ACAAAAG: 0.011 ACAAAAT: 0.033...
...

Probabilistic gene finder using a 1st order model on codons
Oops (object-oriented programs) example 1: simulating a population of RNA organisms

an RNA ‘organism’

AGAAAAAACAA

Fitness (probability of reproducing) depends on number of secondary structure pairing interactions.
Selection and reproduction over a series of generations

- 1/3 of sequences with most pairing interactions selected to form “breeding population”
- Sample with replacement to obtain parent sequences
- Replicate these with mutation to form next generation
def sim(seq_len, pop_size, num_gens):
    """Evolve RNA strings over num_gens generations."""
    # get initial population
    pop = initial_pop(pop_size, seq_len)
    print('Initial population fitness', mean_fitness(pop))

    # evolve...
    for i in range(num_gens):
        pop = next_gen(pop)

    # print mean fitness of final population
    print('Final population fitness', mean_fitness(pop))

    pop.sort(reverse=True)
    return pop
Getting the next generation

```python
import random

def next_gen(pop):
    """Given a population, find most fit 1/3 and use these to reproduce next generation."""
    # find most fit 1/3
    pop.sort(reverse=True)  # sort high to low
    breed = pop[:int(len(pop)/3)]

    # breed
    new_pop = []
    for i in range(len(pop)):
        parent = random.choice(breed)
        new_pop.append(parent.replicate())

    return new_pop
```

for this to work, class rnaOrg must have __eq__ and __lt__ methods

and rnaOrg must have replicate method
class rnaOrg:

def __init__(self, seq):
    """An RNA organism.""
    self.seq = seq

def get_fitness(self):
    """Return total number of pairing interactions in our sequence."""

def __repr__(self):

def __eq__(self, other):
    """Return True if this organism is equally fit as other organism."""

def __lt__(self, other):
    """Return True if this organism is less fit than other organism."""

def replicate(self): (stretch goal)
    """Create a new organism with a potentially mutated genome."""

    • How should this class represent fitness?
    • Assume mfold5(seq, {}) is available

    Worksheet

    • Assume you can use random
    • Compute probability, then mutate if p < MUTPROB
class rnaOrg:

def __init__(self, seq):
    """An RNA organism.""
    self.seq = seq
    self.fitness = self.get_fitness()

def get_fitness(self):
    """Return total number of pairing interactions in our sequence.""
    return mfold5(self.seq, {})

def __repr__(self):
    return str(self.fitness) + " " + self.seq

def __eq__(self, other):
    """Return True if this organism is equally fit as other organism.""
    return self.fitness == other.fitness

def __lt__(self, other):
    """Return True if this organism is less fit than other organism.""
    return self.fitness < other.fitness

def replicate(self):
    """Create a new organism with a potentially mutated genome.""
    new_seq = []
    for base in self.seq:
        if random.random() < MUTPROB:
            new_seq.append(random.choice(['A', 'U', 'C', 'G']))
        else:
            new_seq.append(base)
    return rnaOrg("".join(new_seq))
Oops example 2: dates

```python
>>> today = Date(11, 15, 2018)
>>> due = Date(11, 20, 2018)
>>> due - today
5
```

```python
class Date:
    def __init__(self, day, month, year):
```

What is that red minus!?
Oops example 2: dates

```python
>>> today = Date(11, 15, 2018)
>>> due = Date(11, 20, 2018)
>>> due - today
5
```

```python
class Date:
    def __init__(self, day, month, year):
    def __sub__(self, other):
        blah, blah, blah
```
Oops example 2: dates

```python
today = Date(11, 15, 2018)
due = Date(11, 20, 2018)
if due > today:
    print("let's watch a movie!")
```

class Date:
    def __init__(self, m, d, y):
        self.month = m
        self.day = d
        self.year = y
    def __sub__(self, other):
    def __gt__(self, other):

>>> d = Date(11, 15, 2018)
Another implementation...

```python
class Date:
    def __init__(self, m, d, y):
        self.days_since_JanFirst1900 = \textit{funky math here!}

    def __sub__(self, other):

    def __gt__(self, other):

>>> d = Date(11, 15, 2018)
```

Why would any sane person want to store the date as the number of days since January 1, 1900?
Converting in and out of an internal representation

class Date:
    def __init__(self, m, d, y):
        self.days_since_JanFirst1900 = \
            self.get_days_since_1900(m, d, y)

    def get_days_since_1900(self, m, d, y):
        funky math here

    def get_month_day_year(self):
        funky math in reverse here

>>> d = Date(11, 15, 2018)
>>> d.get_month_day_year ()
(11, 15, 2018)
Date “Abstraction”

Date

```
__init__(self, month, day, year)
get_days_since_1900(self, m, d, y)
get_month_day_year(self)
==, >, <, >=, <=, +, -
```
A final oops example: protein-protein interaction networks
Some input data

edges = [
    (gene4634, gene2542),
    (gene2351, gene3807),
    (gene207, gene2331),
    (gene2180, gene4867),
    ...
    ...
    ...
    (gene4224, gene2073),
    (gene4128, gene1902),
    (gene785, gene4093),
    (gene3879, gene1734),
    (gene4906, gene2255),
]
def is_connected(gene1, gene2, edges):
    """Return True if gene1 and gene2 are connected in edges."""
def is_connected(gene1, gene2, edges):
    """Return True if gene1 and gene2 are connected in edges."""

    for geneA, geneB in edges:
        if geneA == gene1 and geneB == gene2:
            return True
        elif geneA == gene2 and geneB == gene1:
            return True

    return False
What if the network is really big and we have a lot of queries?

```python
def query_edges(qedges, edges):
    """Look for qedges in edges. Return list of those present."""
    present = []
    for q1, q2 in qedges:
        if is_connected(q1, q2, edges):
            present.append((q1, q2))
    return present
```

The technical term is hairball.

A cancer related protein-protein interaction network

A network class

class Network:

    def __init__(self, edges):
        """Protein-protein interaction network."""
        self.adj_list = {}
        for geneA, geneB in edges:
            self.add_edge(geneA, geneB)

    def add_edge(self, geneA, geneB):
        """Add edge to network."""
        if geneA not in self.adj_list:
            self.adj_list[geneA] = []
            self.adj_list[geneA].append(geneB)

        if geneB not in self.adj_list:
            self.adj_list[geneB] = []
            self.adj_list[geneB].append(geneA)

adj_list is an attribute that stores the network
- keys are genes (proteins)
- values are list of other genes a given gene is connected to
Write an `is_connected` method for this Network class.

```python
def is_connected(self, gene1, gene2):
    """Return True if gene1 and gene2 are connected."""
```
def is_connected(self, gene1, gene2):
    """Return True if gene1 and gene2 are connected."""
    if gene1 in self.adj_list:
        if gene2 in self.adj_list[gene1]:
            return True

    return False
Try the network version out...

def query_edges_network(qedges, network):
    """Look for qedges in edges. Return list of those present."""
    present = []
    for q1, q2 in qedges:
        if network.is_connected(q1, q2):
            present.append((q1, q2))
    return present

>>> network = Network(edges)
>>> query_edges_network(query_edges, network)
See you in lab