predator-prey simulations

1. Hopping Frogs
   - an object oriented model of a frog
   - animating frogs with threads
2. Frogs on Canvas
   - a GUI for hopping frogs
   - stopping and restarting threads
3. Flying Birds
   - an object oriented model of a bird
   - defining a pond of frogs
   - giving birds access to the swamp

MCS 260 Lecture 36
Introduction to Computer Science
Jan Verschelde, 11 April 2016

predator-prey simulations

1. Hopping Frogs
   - an object oriented model of a frog
   - animating frogs with threads
2. Frogs on Canvas
   - a GUI for hopping frogs
   - stopping and restarting threads
3. Flying Birds
   - an object oriented model of a bird
   - defining a pond of frogs
   - giving birds access to the swamp

Modeling Frogs

in object oriented fashion

Imagine frogs happily hopping around in a pond ...

Functional attributes in the class `Frog`:

- `hop` : wait a bit and hop to next spot,
- `run` : do some fixed number of hops.

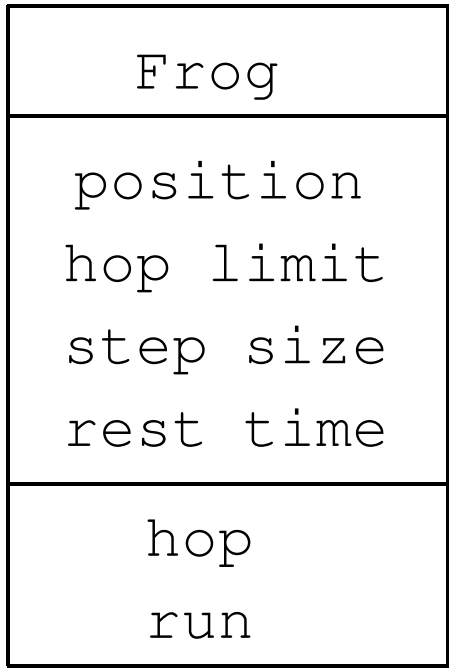
Data attributes in the class `Frog`:

- `position` : coordinates (x, y) for its location,
- `hop limit` : number of hops done by the frog,
- `step size` : largest step size a frog can do,
- `rest time` : time while frog rests between steps.

Adding some randomness makes it more interesting.

The Class Diagram

In the Unified Modeling Language (UML), we draw



script class_frog.py

```
from time import sleep
from random import randint

class Frog(object):
    """
    Object oriented model of a frog.
    """
    def __init__(self, n, x, y, h, d, m):
        """
        A frog with name n at (x,y)
        makes m hops of size at most h,
        resting at most d seconds.
        """
        self.name = n
        self.position = (x, y)
        self.hop_limit = m
        self.step_size = h
        self.rest_time = d
```

functional attributes

```
def hop(self):
    """
    The frog waits a bit before hopping.
    """
    name = self.name
    pos = self.position
    rnd = randint(0, self.rest_time)
    print(name + ' at ' + str(pos) \
      + ' waits ' + str(rnd) + ' seconds')
    sleep(rnd)
    hopstep = self.step_size
    (xps, yps) = self.position
    nxp = xps + randint(-1, +1)*hopstep
    nyp = yps + randint(-1, +1)*hopstep
    self.position = (nxp, nyp)
    print(name + ' hops to ' + str((nxp, nyp)))
```

functions run and main

```
    def run(self):
        """
        The frog does as many hops
        as the value set by self.hop_limit.
        """
        while self.hop_limit > 0:
            self.hop()
            self.hop_limit = self.hop_limit - 1

def main():
    """
    Kermit our hero hops around!
    """
    print('Kermit the frog is born ...')
    k = Frog('Kermit', +10, +10, 5, 2, 4)
    print('Kermit starts hopping ...')
    k.run()
```

predator-prey simulations

1. Hopping Frogs
   - an object oriented model of a frog
   - animating frogs with threads
2. Frogs on Canvas
   - a GUI for hopping frogs
   - stopping and restarting threads
3. Flying Birds
   - an object oriented model of a bird
   - defining a pond of frogs
   - giving birds access to the swamp

multithreading in Python

many frogs hop independently

Most processors have multiple cores, allowing for parallel execution in real time, speeding up calculations.

Even on one core, the operating system runs many threads.

Python provides the Thread class in the threading module.

In addition to the `__init__` we override the definition of `run` when defining a class, inheriting from `Thread`.

For `t`, an instance of the inherited Thread class:
`t.start()` executes the defined `run` method.

$\rightarrow$ multiple threads run simultaneously

script class_threadfrog.py

```
from threading import Thread
from class_frog import Frog

class ThreadFrog(Thread, Frog):
    """
    Exports hopping frogs as threads.
    """
    def __init__(self, n, x, y, h, d, m):
        """
        A frog with name n at (x,y)
        makes m hops of size at most h,
        resting at most d seconds.
        """
        Thread.__init__(self, name=n)
        Frog.__init__(self, n, x, y, h, d, m)
```

We inherit from `Thread` and `Frog`.

script `class_threadfrog.py` continued ...

```
def run(self):
    """
    The frog does as many moves
    as the value set by self.moves.
    """
    while self.hop_limit > 0:
        self.hop()
        self.hop_limit = self.hop_limit - 1
```

The `hop()` method is defined in `class_frog.py`,
as a method of the class `Frog`.

the `main()` in `class_script`

```
def main():
    """
    Runs a simple test on hopping frogs.
    """
    print('creating two frogs: a and b')
    ann = ThreadFrog('a', +10, +10, 5, 2, 4)
    bob = ThreadFrog('b', -10, -10, 15, 5, 3)
    print('starting the hopping ...')
    ann.start()
    bob.start()
    print('waiting for frogs to finish ...')
    ann.join()
    bob.join()

if __name__ == "__main__":
    main()
```

predator-prey simulations

1. Hopping Frogs
   - an object oriented model of a frog
   - animating frogs with threads
2. Frogs on Canvas
   - a GUI for hopping frogs
   - stopping and restarting threads
3. Flying Birds
   - an object oriented model of a bird
   - defining a pond of frogs
   - giving birds access to the swamp

Frogs on Canvas

a GUI for hopping frogs

In our GUI, the canvas will be the pond.

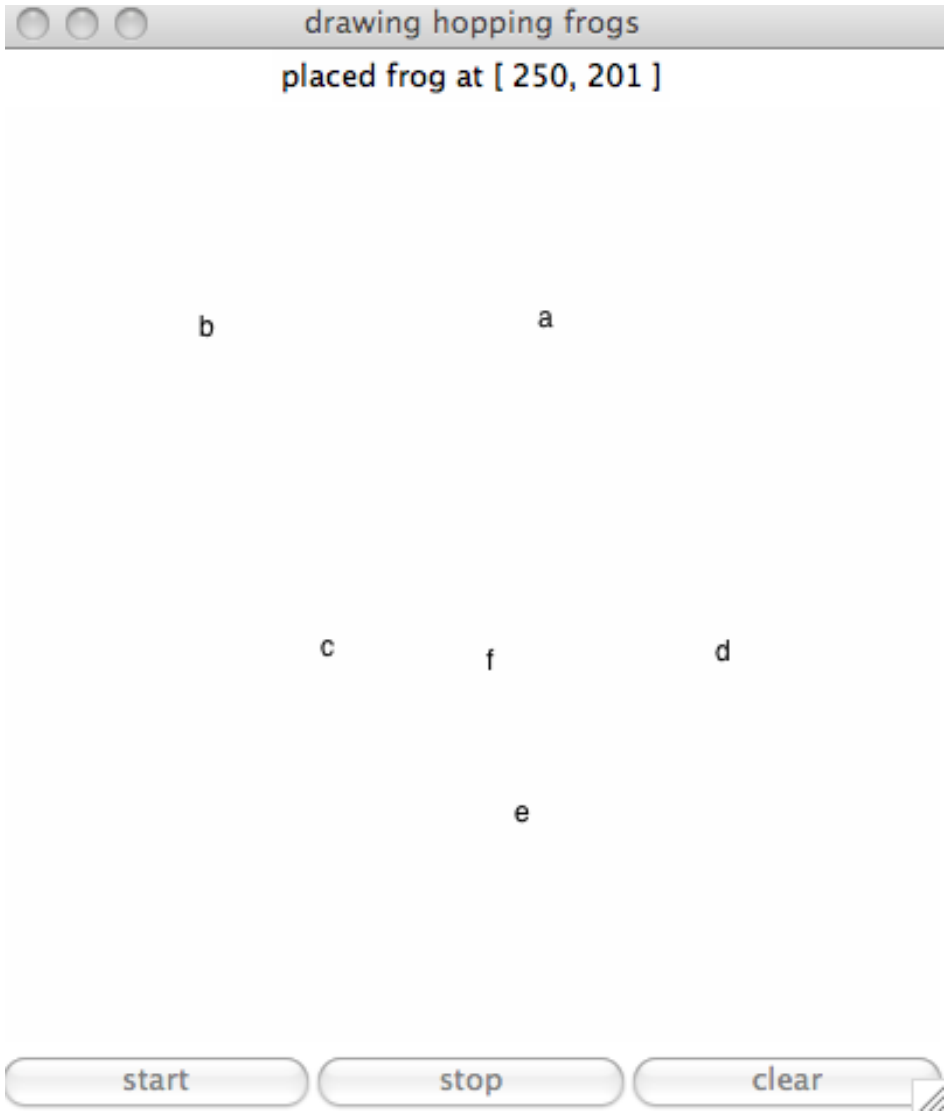
The user can place frogs in the pond with the mouse.

Three buttons:

- `start` : to start animating the hopping of frogs,
- `stop` : to stop the animation for adding frogs,
- `clear` : to clear the canvas after killing frogs.

The names of the frogs are letters `'a'`, `'b'`, `'c'`, ...
used as text to mark the frogs on canvas.

running the GUI



data attributes

```
from tkinter import Tk, StringVar, Label, Canvas
from tkinter import Button, W, E, ALL
from class_threadfrog import ThreadFrog

class DrawFrogs(object):
    """
    GUI to hopping frogs on canvas
    """
    def __init__(self, wdw, d):
        """
        Defines a square canvas of d pixels, label,
        start, stop, and clear buttons.
        """
        wdw.title("drawing hopping frogs")
        self.frogs = []
        self.gohop = False
        self.dim = d
        self.msg = StringVar()
```

label, canvas, bindings, buttons

```
self.msg.set("put mouse inside box to place frogs")
self.lab = Label(wdw, textvariable=self.msg)
self.lab.grid(row=0, column=0, columnspan=3)
self.cnv = Canvas(wdw, width=d, height=d, bg='white')
self.cnv.grid(row=1, columnspan=3)
self.cnv.bind("<Button-1>", self.button_pressed)
self.cnv.bind("<ButtonRelease-1>", self.button_released)
self.bt0 = Button(wdw, text='start', command=self.start)
self.bt0.grid(row=2, column=0, sticky=W+E)
self.bt1 = Button(wdw, text='stop', command=self.stop)
self.bt1.grid(row=2, column=1, sticky=W+E)
self.bt2 = Button(wdw, text='clear', command=self.clear)
self.bt2.grid(row=2, column=2, sticky=W+E)
```

defining mouse bindings

```
def button_pressed(self, event):
    """
    Displays coordinates of button pressed.
    """
    self.msg.set("currently at [ " + \
        str(event.x) + ", " + str(event.y) + " ]" +\
        " release to place frog")

def button_released(self, event):
    """
    At release of button, a frog is created at
    the current location of the mouse pointer.
    """
    self.msg.set("placed frog at [ " + \
        str(event.x) + ", " + str(event.y) + " ]")
    self.new_frog(event.x, event.y)
    self.draw_frogs()
```

a new frog

```
def new_frog(self, xps, yps):
    """
    Defines a new frog with coordinates
    given in (xps, yps) via the mouse.
    """
    nbr = len(self.frogs)
    name = chr(ord('a')+nbr)
    frog = ThreadFrog(name, xps, yps, 20, 5, 100)
    self.frogs.append(frog)
```

drawing frogs

```
def draw_frogs(self):
    """
    Draws frogs on canvas, eventually
    after fixing their coordinates.
    """
    dim = self.dim
    for frog in self.frogs:
        (xps, yps) = frog.position
        name = frog.getName()
        self.cnv.delete(name)
        if xps < 0:
            xps = dim - xps
        if yps < 0:
            yps = dim - yps
        if xps > dim:
            xps = xps - dim
        if yps > dim:
            yps = yps - dim
        frog.position = (xps, yps)
        self.cnv.create_text(xps, yps, text=name, tags=name)
```

starting the animation

```
def start(self):
    """
    Starts all frogs and the animation.
    """
    self.gohop = True
    for frog in self.frogs:
        frog.start()
    while self.gohop:
        self.draw_frogs()
        self.cnv.after(10)
        self.cnv.update()
```

predator-prey simulations

1. Hopping Frogs
   - an object oriented model of a frog
   - animating frogs with threads
2. Frogs on Canvas
   - a GUI for hopping frogs
   - stopping and restarting threads
3. Flying Birds
   - an object oriented model of a bird
   - defining a pond of frogs
   - giving birds access to the swamp

stopping an animation

We cannot halt threads,
but we kill frogs by setting their `hop_limit` to zero.

Setting `hop_limit` stops the loop in `run`
and then we create a new frog with the same data.

```
def clear(self):
    """
    Deletes all frogs from canvas.
    """
    print('killing all frogs ...')
    for frog in self.frogs:
        frog.hop_limit = 0
    while len(self.frogs) > 0:
        self.frogs.pop(0)
    self.cnv.delete(ALL) # cleaning canvas
```

stopping and restarting

```
def stop(self):
    """
    Sets hop limits of frogs to zero.
    """
    self.gohop = False
    print('setting moves of frogs to zero ...')
    for frog in self.frogs:
        frog.hop_limit = 0
    print('resetting all frogs ...')
    newfrogs = []
    while len(self.frogs) > 0:
        frog = self.frogs.pop(0)
        pos = frog.position
        step = frog.step_size
        rest = frog.rest_time
        newf = ThreadFrog(frog.getName(), pos[0], pos[1], \
            step, rest, 100)
        newfrogs.append(newf)
    self.frogs = newfrogs
```

predator-prey simulations

1. Hopping Frogs
   - an object oriented model of a frog
   - animating frogs with threads
2. Frogs on Canvas
   - a GUI for hopping frogs
   - stopping and restarting threads
3. Flying Birds
   - an object oriented model of a bird
   - defining a pond of frogs
   - giving birds access to the swamp

Modeling Birds

in object oriented fashion

Birds fly over the pond, preying on frogs.

Functional attributes for an object of the class `Bird`:

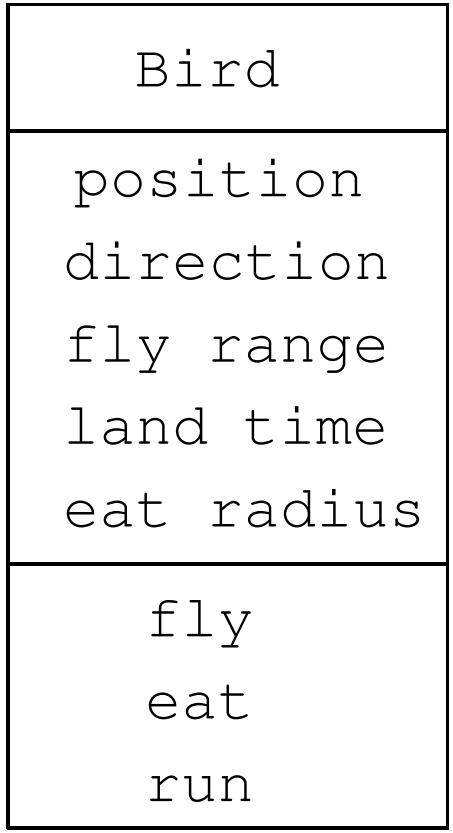
- `fly` : fly in some direction and then land,
- `eat` : eat closest frog within a certain radius,
- `run` : do `fly()` and `eat()` a fixed number of times.

Data attributes in the class `Bird`:

- `position` : coordinates (x, y) for its location,
- `direction` : tuple (d_x, d_y) defines flying direction,
- `fly range` : how many times a bird flies and lands,
- `land time` : time spent on land between air trips,
- `eat radius` : distance for closest frog to be eaten.

The Class Diagram

In the Unified Modeling Language (UML), we draw



script flying_birds.py

```
class Bird(Thread):
    """
    Exports flying birds as threads.
    """
    def __init__(self, n, p, d, m, s, r):
        """
        A bird with name n at position p
        flies m times in the direction d,
        spending at most s seconds on land,
        eating frogs within radius r.
        """
        Thread.__init__(self, name=n)
        self.position = p
        self.direction = d
        self.fly_range = m
        self.land_time = s
        self.eat_radius = r
```

birds fly

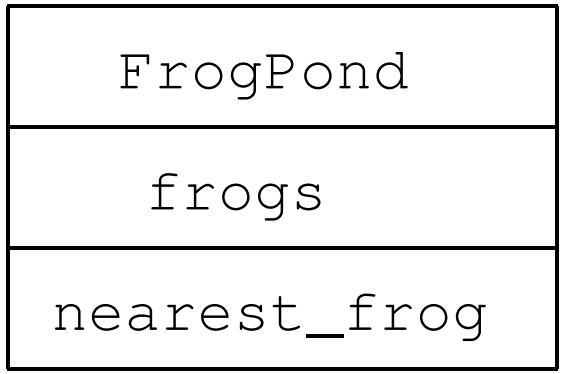
```
def fly(self):
    """
    The bird waits a bit before flying.
    """
    name = self.getName()
    pos = self.position
    tsec = randint(0, self.land_time)
    print(name + ' at ' + str(pos) \
        + ' lands ' + str(tsec) + ' seconds')
    sleep(tsec/2)
    self.eat()
    sleep(tsec/2)
    (xps, yps) = self.position
    nxps = xps + self.direction[0]
    nyps = yps + self.direction[1]
    self.position = (nxps, nyps)
    print(name + ' flies to ' + str((nxps, nyps)))
```

predator-prey simulations

1. Hopping Frogs
   - an object oriented model of a frog
   - animating frogs with threads
2. Frogs on Canvas
   - a GUI for hopping frogs
   - stopping and restarting threads
3. Flying Birds
   - an object oriented model of a bird
   - defining a pond of frogs
   - giving birds access to the swamp

finding the closest frog

To define the `eat()` method of the class `Bird`,
we need to compute the closest frog.



script frog_pond.py

```
from math import sqrt
from class_threadfrog import ThreadFrog

class FrogPond(object):
    """
    A pond keeps the frogs and allows
    to locate the frogs for predators.
    """
    def __init__(self, F):
        """
        F is a list of frogs,
        which may be empty at first.
        """
        self.frogs = F
```

computing the distance

With `@staticmethod` we indicate a method without `self`:

```
@staticmethod
def distance(pnt, qnt):
    """
    Returns the distance between points pnt and qnt.
    """
    sqrdif = (pnt[0] - qnt[0])**2 + (pnt[1] - qnt[1])**2
    return sqrt(sqrdif)
```

The method is then called as `FrogPond.distance()`.

A static method is similar to a class wide data attribute.

computing the nearest frog

```
def nearest_frog(self, pos):
    """
    On input in pos is a tuple (x, y)
    with numerical coordinates in the plane.
    Returns (-1, None, -1) for an empty pond,
    or the tuple (d, f, i) where d is the
    distance of pos to the nearest frog
    at position i in the list of frogs.
    """
    if len(self.frogs) == 0:
        return (-1, None, -1)
    else:
        k = 0
        mindis = FrogPond.distance(self.frogs[k].position, pos)
        for i in range(1, len(self.frogs)):
            dis = FrogPond.distance(self.frogs[i].position, pos)
            if dis < mindis:
                (k, mindis) = (i, dis)
        return (mindis, self.frogs[k], k)
```

testing the class

```
def main():
    """
    Runs a test on the FrogPond class.
    """
    print('swamp with two frogs: a and b')
    ann = ThreadFrog('a', +10, +10, 5, 2, 4)
    bob = ThreadFrog('b', -10, -10, 15, 5, 3)
    swamp = FrogPond([ann, bob])
    while True:
        posraw = input("Give a tuple of coordinates : ")
        pos = eval(posraw)
        near = swamp.nearest_frog(pos)
        if near[0] != -1:
            print('nearest frog to', pos, 'is', \
            near[1].getName(), 'at distance', \
            near[0], 'at position', near[2])
        ans = input("more locations ? (y/n) ")
        if ans != 'y':
            break
```

predator-prey simulations

1. Hopping Frogs
   - an object oriented model of a frog
   - animating frogs with threads
2. Frogs on Canvas
   - a GUI for hopping frogs
   - stopping and restarting threads
3. Flying Birds
   - an object oriented model of a bird
   - defining a pond of frogs
   - giving birds access to the swamp

at start of `flying_birds.py`

```
from class_threadfrog import ThreadFrog
from frog_pond import FrogPond

print('swamp with four frogs')
ANN = ThreadFrog('ann', +10, +10, 2, 5, 20)
BOB = ThreadFrog('bob', +10, -10, 2, 5, 20)
CINDY = ThreadFrog('cindy', -10, +10, 2, 5, 20)
DAVE = ThreadFrog('dave', -10, -10, 2, 5, 20)
SWAMP = FrogPond([ANN, BOB, CINDY, DAVE])

class Bird(Thread):
    """
    Exports flying birds as threads.
    """
```

method `eat()`

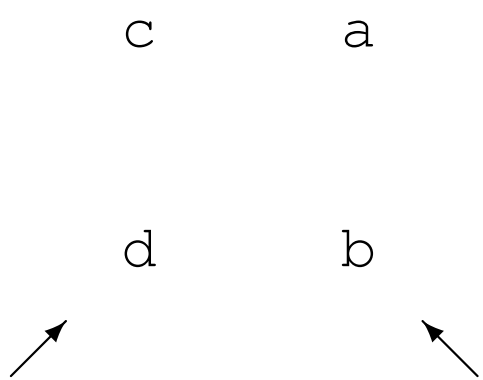
```
def eat(self):
    """
    Finds closest frog to the bird
    and eats it if within its radius.
    """
    name = self.getName()
    prey = SWAMP.nearest_frog(self.position)
    if prey[0] != -1:
        prn = prey[1].getName()
        spr = 'nearest frog to ' + name \
            + ' is ' + prn \
            + ' at distance ' + ('%.2f' % prey[0])
        if prey[0] > self.eat_radius:
            print(spr)
        else:
            print(spr + ' => ' + prn + ' gobbled up')
            prey[1].hop_limit = 0
            del SWAMP.frogs[prey[2]]
```

overriding the `run`

```
def run(self):
    """
    The bird flies as many times
    as the value set by self.fly_range.
    """
    while self.fly_range > 0:
        self.fly()
        self.fly_range = self.fly_range - 1
```

testing the simulation

We have a swamp with four frogs,
initially placed as below:



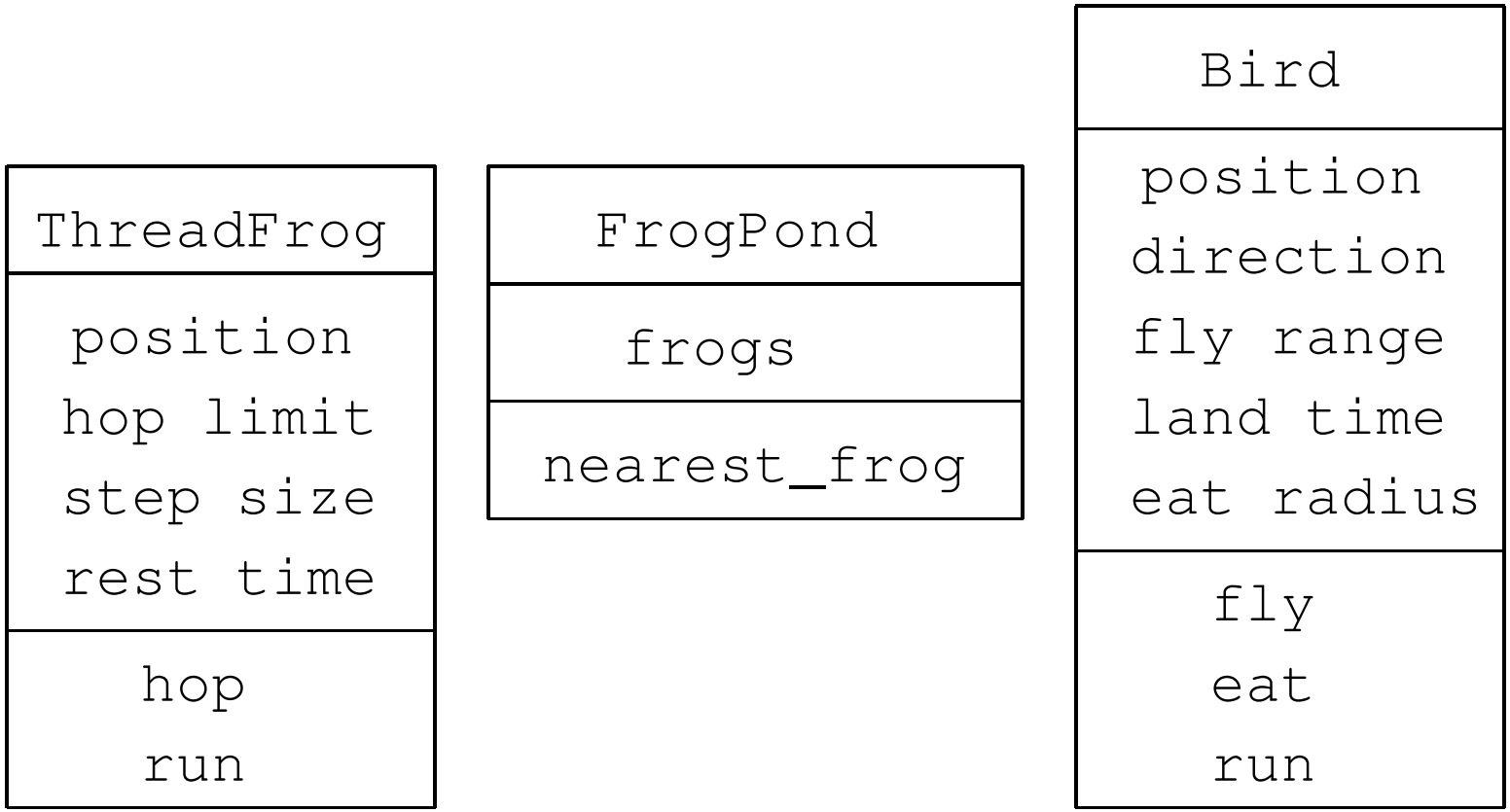
and two birds flying into the pond,
with directions drawn by arrows above.

As they lie straight in the path of the birds,
frogs `b` and `d` are easy preys,
while `a` and `c` may hop away, escaping the birds.

the main program

```
def main():
    """
    Runs a simple test on flying birds.
    """
    print('creating two birds: A and B')
    (apos, adir) = ((-15, -15), (+1, +1))
    (bpos, bdir) = ((+15, -15), (-1, +1))
    apred = Bird('A', apos, adir, 30, 4, 5)
    bpred = Bird('B', bpos, bdir, 30, 4, 5)
    print('frogs start life in swamp ...')
    for frog in SWAMP.frogs:
        frog.start()
    print('birds start to fly ...')
    apred.start()
    bpred.start()
    apred.join()
    bpred.join()
```

Three Classes



The class `FrogPond` depends on the class `ThreadFrog`.

The `eat()` method in `Bird` depends on `ThreadFrog` and on an instance of `FrogPond`.

Summary and Assignments

Read chapter 11 of *Python Programming*.

Assignments:

1. Extend the `Frog` class with a method `breed()`. When two frogs are within a certain distance from each other, new frogs are born.
2. Adjust the direction of the bird in the `eat()` method so that the bird flies towards the closest frog.
3. Change the class `Bird` so the simulation runs from a script not in the same file as the definition of the class.
4. Add flying birds to the GUI to place frogs. Represent the birds by circles with radius equal to their eat radius so the user can observe when a frog is gobbled up.