Threads
Multi-threaded programs

- Multiple simultaneous paths of execution
  - Seemingly at once (single core)
  - Actually at the same time (multiple cores)

- Why?
  - Get work done faster (on multi-core machines)
  - Simplify code by splitting up work into parts
  - Remain responsive to user input

- Threads in Python
  - Creating, starting, sleeping
  - Unpredictability
  - Debugging
A Single-Threaded Program: Single Core

```python
class Animal:
    def __init__(self, image='', audio=''):  # Animal.py
        self.image = image
        self.audio = audio
    def show(self):
        StdDraw.picture(picture.Picture(self.image), 0.5, 0.5)
        StdAudio.playFile(self.audio)

map["dog"] = Animal("dog.jpg", "dog")
map["cat"] = Animal("cat.jpg", "cat")
map["cow"] = Animal("cow.jpg", "cow")

while True:
    name = input("Enter animal: ")
    animal = map[name]
    if animal != None:
        animal.show()
    StdDraw.show(100)
```

A single core computer.
A Single-Threaded Program: Multi-Core

```python
class Animal:
    def __init__(self, image='', audio=''):  
        self.image = image  
        self.audio = audio

    def show(self):
        StdDraw.picture(picture.Picture(self.image), 0.5, 0.5)  
        StdAudio.playFile(self.audio)

map['dog'] = Animal('dog.jpg', 'dog')
map['cat'] = Animal('cat.jpg', 'cat')
map['cow'] = Animal('cow.jpg', 'cow')

while True:
    StdDraw.clear()

    name = input('Enter animal: ')

    animal = map[name]

    if animal != None:
        animal.show()

    StdDraw.show(100)
```

A multi-core computer.
Multi-Threaded Animals

- **New “magic” program: AnimalMapDeluxe.py**
  - Random frogs!
  - Frogs appear every second
  - User can make requests:
    - "dog", "cat", "cow"
    - Even while frog is appearing
Creating and Starting a Thread

- **Thread**
  - A separate *path of execution*
    - Also: the name of a class in the Python threading library
  - Program creates an object of type `Thread`

- **Creating and starting a thread:**
  
  ```python
  import threading
  var = threading.Thread()
  var.start()
  ```

  Simple, but doesn't actually do anything:
  - Thread is born
  - Thread dies
  - End of story
Making Work for a Thread

- Thread constructor can take parameters
  - Can take the name of a function as its “target”
  - Can take an object and one of its methods as “target”
  - Can also pass in arguments to those functions and methods through “args” parameter

```python
def BlastOff ():
    for i in range(10, 0, -1):
        print(i, end=" ")
    print("BLAST OFF!")
```
import threading

def BlastOff ():
    for i in range(10, 0, -1):
        print(i, end= " ")
    print("BLAST OFF!")

if __name__ == "__main__":
    print("prepare for launch")
    thread = threading.Thread(target=BlastOff)
    thread.start()
    print("done with launch")

% python Launch.py
prepare for launch
done with launch
10 9 8 7 6 5 4 3 2 1 BLAST OFF!
import threading

def BlastOff ():
    for i in range(10, 0, -1):
        print(i, end=" ")
    print("BLAST OFF!")

if __name__ == "__main__":
    print("prepare for launch")
    thread = threading.Thread(target=BlastOff)
    thread.start()
    print("done with launch")

% python Launch.py
prepare for launch
done with launch
10 9 8 7 6 5 4 3 2 1 BLAST OFF!
import threading

def BlastOff ():
    for i in range(10, 0, -1):
        print(i, end=" ")
    print("BLAST OFF!")

if __name__ == "__main__":
    print("prepare for launch")
    thread = threading.Thread(target=BlastOff)
    thread.start()
    print("done with launch")

% python Launch.py
prepare for launch
10 done with launch
9 8 7 6 5 4 3 2 1 BLAST OFF!
import threading

def BlastOff ():
    for i in range(10, 0, -1):
        print(i, end=" ")
    print("BLAST OFF!")

if __name__ == "__main__":
    print("prepare for launch")
    thread = threading.Thread(target=BlastOff)
    thread.start()
    print("done with launch")

% python Launch.py
prepare for launch
10 9 done with launch
8 7 6 5 4 3 2 1 BLAST OFF!
Thread States: Startup

```
t = threading.Thread(target=r)
t.start()
```

- **NEW**
  - "I'm waiting to get started."
  - Just a normal object on the heap

- **RUNNABLE**
  - "I'm good to go!"
  - Waiting to be selected by the thread scheduler

- **RUNNING**
  - "Can I supersize that for you?"
  - Actually running on the CPU
Thread States

RUNNABLE

RUNNING

BLOCKED

Sent to a temporary non-runnable state, until it can become runnable again.

sleeping, waiting for another thread to finish, waiting for data to be available on the stream, waiting for an object’s lock...
Launching Multiple Threads

- **Main program: launch > 1 threads**
  - Split work into multiple parts

```python
import threading
import concurrent.futures
import sys

def BlastOff():
    for i in range(10, 0, -1):
        print(str(i) + " ", end="")
    print("BLAST OFF!")

if __name__ == "__main__":
    N = int(sys.argv[1])

    print("prepare for multi launch")
    threads = list()
    for index in range(N):
        x = threading.Thread(target=BlastOff)
        threads.append(x)
        x.start()
    print("done with launch")

% python MultiLaunch.py 3
prepare for multi launch
10 10 10 done launching
9 9 8 8 7 8 6 7 5 7 4 6 3 6 2 5 1 5 BLAST OFF!
4 4 3 3 2 2 1 BLAST OFF!
1 BLAST OFF!
```
Important Thread Tricks

- **Main thread can wait for workers**
  - e.g. Merge results from workers
  - Call `join()` on the thread object

- **Passing data to/from a worker**
  - `Thread()` constructor is passed a `Runnable` object
  - Add any instance variables / methods you want
    - Input: send as parameters to `constructor`
    - Output: add some `get data method(s)`
  - But: you must keep track of object sent to `Thread()`
```python
class FibWorker:
    def __init__(self, num):
        self.num = num
        self.result = 0
    def getResult(self):
        return self.result
    def run(self):
        self.result = self.fib(self.num)
    def fib(self, n):
        if n == 0:
            return 0
        if n == 1:
            return 1
        return self.fib(n-1) + self.fib(n-2)
```

Get the input, what we'll calculate once somebody says go!

Somebody getting the output. Should only be called after the thread is known to be done.

Method that we're using when we call .start() on a Thread that has been passed this object and method.
from FibWorker import FibWorker
import threading
import sys

if __name__ == "__main__":
    threads = list()
    workers = list()
    for i in range(1, len(sys.argv)):
        workers.append(FibWorker(int(sys.argv[i])))
        threads.append(threading.Thread(target=workers[i-1].run()))
        threads[i-1].start()
    for i in range(1, len(sys.argv)):
        threads[i-1].join()
    print("fib(" + sys.argv[i] + ") = ", end = "")
    print(workers[i-1].getResult())

We are keeping track of two parallel arrays, one for threads and one for worker objects.

Setup worker with its job.

Once a thread is done, we know it has a good output value.
Sleeping

• Making a thread take a nap
  ○ Specify **nap time in seconds**
    ▫ Guaranteed to sleep at least this long
      ○ Maybe longer though
    ▫ Allows other threads to enter running state
    ▫ **Polite behavior** when you've got nothing to do
    ▫ `time.sleep(sec)`

```
while not StdDraw.hasNextKeyTyped():
    # Burns an entire core to do nothing!
    char ch = StdDraw.nextKeyTyped()
```

```python
import time

while not StdDraw.hasNextKeyTyped():
    time.sleep(.001)
    char ch = StdDraw.nextKeyTyped()
```
**Naming Threads**

- **Threads can be given a name**
  - Helpful for debugging
  - “name” parameter to `Thread()` constructor

```python
if __name__ == "__main__":
    N = int(sys.argv[1])

    print("prepare for multi launch")
    threads = list()
    for index in range(N):
        x = threading.Thread(name='B'+str(index), target=BlastOffSleep)
        threads.append(x)
        x.start()
    print("done with launch")
```
def BlastOffSleep ():
    myName = threading.currentThread().getName()
    for i in range(10, 0, -1):
        print(str(i) + "(" + myName + ")", end="")
        time.sleep(1)
    print("BLAST OFF!")

% python MultiLaunchSleep.py 3
prepare for multi launch
10(B0) done launching
10(B2) 10(B1) 9(B0) 9(B1) 9(B2) 8(B0) 8(B1) 8(B2) 7(B0) 7(B1) 7(B2)
6(B1) 6(B0) 6(B2) 5(B0) 5(B2) 5(B1) 4(B0) 4(B1) 4(B2) 3(B0) 3(B1)
3(B2) 2(B0) 2(B1) 2(B2) 1(B0) 1(B1) 1(B2) BLAST OFF! (B0)
BLAST OFF! (B1)
BLAST OFF! (B2)
Summary

- **Python Thread**
  - Multiple *simultaneous paths of execution*
    - Really simultaneous (multiple cores)
    - Simply seem that way (single core)

- **Important thread skills:**
  - Implementing a worker class
  - Starting a thread
  - Making a thread sleep
  - Waiting for a thread to finish
  - Getting input to a thread
  - Getting output from a thread
Your Turn

- **Create a function that:**
  - Draws something using `StdDraw` in **unit box**
  - Sleeps at least 500ms
  - Changes something about the drawing
  - Repeats forever
  - Don't worry about erasing
    - Don't call `StdDraw.clear()`
  - I'll integrate into my ThreadZoo program and run this next lecture class

- Open Moodle, go to CSCI 136, Section 11
- Open the dropbox for today – Activity 5 - Threads
- Drag and drop your program file to the Moodle dropbox
- You get: 1 point if you turn in something, 2 points if you turn in something that is correct.