Lecture 25

Coroutines
Announcements for This Lecture

Assignment & Lab

• A6 is not graded yet
  ▪ Done early next week
  ▪ Survey still open today

• A7 due **Tues, Dec. 7**
  • Extensions are possible
  • Contact your lab instructor

  ▪ Lab Today: Office Hours
    • Get help on A7 aliens
    • Anyone can go to any lab

Optional Videos

• **ALL** all are now posted
  ▪ **Lesson 29** for today
  ▪ **Lesson 30** is the last
Animating Objects

• **Naïve** animations are easy
  - Look at the key input right now
  - Move the objects based on the keys
  - Redraw the moved objects

• **Timed** animations are harder
  - Press a key to start the animation
  - Animation continues for X seconds
  - Animation stops automatically when done

```
animate1.py
```

```
animate2.py
```
def _animate_turn(self, dt):
    """Animates a rotation of the image over SPEED seconds"""
    # Compute degrees per second
    steps = (self._fangle - self._sangle) / SPEED
    amount = steps * FRAME_RATE
    # Update the angle
    self.image.angle = self.image.angle + amount
    # If we go to far, clamp and stop animating
    if abs(self.image.angle - self._sangle) >= 90:
        self.image.angle = self._fangle
        self._animating = False
def _animate_turn(self, dt):
    """Animates a rotation of the image over SPEED seconds"""
    # Compute degrees per second
    steps = (self._fangle - self._sangle) / SPEED
    amount = steps * FRAME_RATE
    # Update the angle
    self.image.angle = self.image.angle + amount
    # If we go too far, clamp and stop animating
    if abs(self.image.angle - self._sangle) >= 90:
        self.image.angle = self._fangle
        self._animating = False
def _animate_turn(self, direction):
    
    """Animates a rotation of the image over SPEED seconds""
    
    sangle = self.image.angle
    fangle = sangle+90 if direction == 'left' else sangle-90
    steps = (fangle-sangle)/ANIMATION_SPEED  # Degrees per second
    animating = True

    while animating:
        amount = steps*FRAME_RATE
        self.image.angle = self.image.angle+amount  # Update the angle
        if abs(self.image.angle-sangle) >= 90:
            self.image.angle = fangle
            animating = False
Wouldn’t a Loop Be Simpler?

def _animate_turn(self, direction):
    """Animates a rotation of the image over SPEED seconds"""
    sangle = self.image.angle
    fangle = sangle+90 if direction == 'left' else sangle-90
    steps = (fangle-sangle)/ANIMATION_SPEED  # Degrees per second
    animating = True
    while animating:
        amount = steps*FRAME_RATE
        self.image.angle = self.image.angle + amount  # Update the angle
        if abs(self.image.angle-sangle) >= 90:
            self.image.angle = fangle
            animating = False

Only Attribute Loop is explicit. Animate until done.
But This is Not Going to Work

• This won’t actually draw anything!
  ▪ This function is a helper to `update()`
  ▪ Keeps running until animation done
  ▪ Method `draw()` only called `at the end`

• Cannot `draw()` inside of `update()`
  ▪ All drawing must be at `same time`
  ▪ What about all the other animations?

• Need some way to “break up” the loop
Doing this With a Bunch of Animations
Doing this With a Bunch of Animations
Doing this With a Bunch of Animations

We need to *multitask* all these animations
What Do We Mean by Multitasking?

Concurrency

- All programs *make progress*
  - Switch between programs
  - Switches are very fast (μs)
- Looks/feels simultaneous

Parallelism

- Programs *run at same time*
  - Each program gets CPU/core
  - No switching between progs
- Actually is simultaneous

Multitasking on old hardware

Multitasking on modern hardware
An Important Distinction

Concurrency

Parallelism

prog 1

prog 2

prog 1

prog 2

Coroutines
# Switching in Currency

<table>
<thead>
<tr>
<th>Preemptive</th>
<th>Cooperative</th>
</tr>
</thead>
</table>
| • Can switch at any time  
  ▪ Even in middle of command!  
  ▪ Cannot prevent switching  
• Very **hard to program** for  
  ▪ Must prepare for anything!  
  ▪ Debugging is a total nightmare  
• Popularized by Unix systems  
  ▪ Many users on one machine  
  ▪ All need “equal” access  | • Only switch at special points  
  ▪ Program specifies when okay  
  ▪ Returns back to this spot  
• Can be easily **abused**  
  ▪ Program never specifies okay  
  ▪ That program hogs machine  
• Popular in early days of GUIs  
  ▪ Okay for main app to hog  
  ▪ No expectation of other apps  |
## Switching in Currency

### Preemptive
- Can switch at any time
  - Even in middle of command!
  - Cannot prevent switching
- Popularized by Unix systems
  - Many users on one machine
  - All need “equal” access
- **Implement with threads**

### Cooperative
- Only switch at special points
  - Program specifies when okay
  - Returns back to this spot
- Popular in early days of GUIs
  - Okay for main app to hog
  - No expectation of other apps
- **Implement with coroutines**
Preemptive Largely Won Out

• Modern OSs moved away from cooperative
  ▪ Windows went preemptive with Windows 95
  ▪ MacOS went preemptive with MacOS X

• Why? The rise of **parallelism**
  ▪ Threads can be concurrent *and* parallel
  ▪ Coroutines are not (easily) parallel

• But threads have **never** gotten easier
  ▪ We have tried for decades (many PhD theses)
  ▪ Still the source of a lot of buggy code
But Coroutines Are Coming Back

• Have figured better ways to parallelize
  ▪ Not as good as threads in general
  ▪ But better/easier for certain applications

• Sometimes explicit coordination is good
  ▪ **Example:** Client-server communication
  ▪ One waits for the other until it responds

• And also relevant to graphical applications
  ▪ They make a lot of animation code easier
  ▪ Used heavily by the Unity game engine
Aside: Subroutine

- A subroutine is a piece of code that
  - Is a set of frequently used instructions
  - Performs a specific task, packaged as a unit
  - Often serves to aid a larger program (routine)
- This sounds just like a function!
  - Not all programming languages have functions
  - This is a generic term that applies to all
- Not a term commonly in use these days
# Subroutines vs Coroutines

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Coroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Runs until completed</td>
<td></td>
</tr>
<tr>
<td>- Invoked by parent routine</td>
<td></td>
</tr>
<tr>
<td>- Runs until reach the end</td>
<td></td>
</tr>
<tr>
<td>- Returns output to parent</td>
<td></td>
</tr>
<tr>
<td>- Just like a function call</td>
<td></td>
</tr>
<tr>
<td>- Parent is “frozen”</td>
<td></td>
</tr>
<tr>
<td>- Subroutine/function runs</td>
<td></td>
</tr>
<tr>
<td>- Parent resumes when done</td>
<td></td>
</tr>
<tr>
<td>- Can stop and start</td>
<td></td>
</tr>
<tr>
<td>- Runs for a little while</td>
<td></td>
</tr>
<tr>
<td>- Returns control to parent</td>
<td></td>
</tr>
<tr>
<td>- And then picks up again</td>
<td></td>
</tr>
<tr>
<td>- <em>Kind of</em> like a generator</td>
<td></td>
</tr>
<tr>
<td>- Starts up at initial call</td>
<td></td>
</tr>
<tr>
<td>- Can yield execution</td>
<td></td>
</tr>
<tr>
<td>- Resumes with full state</td>
<td></td>
</tr>
</tbody>
</table>
Subroutines vs Coroutines

Subroutine

Program 1
Frozen

Program 2

Coroutine

Program 1
Frozen

Frozen

Frozen

Program 2

Frozen

Frozen

Frozen
Subroutines vs Coroutines

Subroutine

Program 1
Call
Return

Frozen

Program 2

Coroutine

Program 1

Frozen

Program 2

Frozen

Next

Yield

Frozen

Next

Yield

Frozen

11/23/21

Coroutines
Subroutines vs Coroutines

### Subroutine

- **Parent**
- **Child**
- **Frozen**
- **call**
- **return**

### Coroutine

- **Parent**
- **Child**
- **Frozen**
- **yield**
- **next**
def _animate_turn(self, direction):
    """Animates a rotation of the image over SPEED seconds""
    sangle = self.image.angle
    fangle = sangle + 90 if direction == 'left' else sangle - 90
    steps = (fangle - sangle) / ANIMATION_SPEED  # Compute degrees per second
    animating = True
    while animating:
        amount = steps * FRAME_RATE
        self.image.angle = self.image.angle + amount  # Update the angle
        if abs(self.image.angle - sangle) >= 90:
            self.image.angle = fangle
            animating = False
        yield  # Pause to draw
```python
def _animate_turn(self, direction):
    """Animates a rotation of the image over SPEED seconds"""
    sangle = self.image.angle
    fangle = sangle+90 if direction == 'left' else sangle-90
    steps = (fangle-sangle)/ANIMATION_SPEED  # Compute degrees per second
    animating = True
    while animating:
        amount = steps*FRAME_RATE
        self.image.angle = self.image.angle+amount  # Update the angle
        if abs(self.image.angle-sangle) >= 90:
            self.image.angle = fangle
            animating = False
    yield  # Pause to draw
```

Add this one line to complete the animation.
def update(self, dt):
    
    """Animates the image.""
    if not self._animator is None:    # Something to animate
        try:
            next(self._animator)    # Step animation forward
        except StopIteration:
            self._animator = None    # Stop animating

    elif self.input.is_key_down('left'):    # Start animation on press
        self._animator = self._animate_turn('left')

    ...

Also Need to Drive The Animation
def update(self, dt):
    
    """Animates the image."""
    if not self._animator is None:
        try:
            next(self._animator)
        except StopIteration:
            self._animator = None
        
    elif self.input.is_key_down('left'):
        self._animator = self._animate_turn('left')

    ...

Ignore input if still animating

Otherwise start animation for given input
def update(self, dt):
    """Animates the image."""
    if not self._animator is None:  # Something to animate
        try:
            next(self._animator)  # Step animation forward
        except StopIteration:
            self._animator = None  # Stop animating
    elif self.input.is_key_down('left'):  # Start animation on press
        self._animator = self._animate_turn('left')
    ...

update is parent of the coroutine
So Are Coroutines Just Generators?

• Generators are an **example** of a coroutine
  ▪ Have parent-child relationship
  ▪ Use `next()` to transfer control to child
  ▪ Child uses `yield` to transfer control back

• But coroutines are a little bit more
  ▪ There is communication **back-and-forth**
  ▪ Yield can give information back to parent
  ▪ But next gives no information to child
So Are Coroutines Just Generators?

• Generators are an example of a coroutine
  ▪ Have parent-child relationship
  ▪ Use `next()` to transfer control to child
  ▪ Child uses `yield` to transfer control back

• But coroutines are a little bit more
  ▪ The
  ▪ Yield
  ▪ Need another command
  ▪ But
Recall: The **yield** Statement

- **Format**: `yield <expression>`
  - Used to produce a value
  - But it **does not stop** the “function”
  - Useful for making iterators
- **But**: These are not normal functions
  - Presence of a yield makes a **generator**
  - Function that returns an iterator
Recall: The **yield** Statement

- **Format:** `yield <expression>`
  - Used to produce a value
  - But it **does not stop** the "function"
  - Useful for making iterators

- **But:** These are not normal functions
  - Presence of a `yield` makes a **generator**
    - Function that returns an iterator

*How do other direction?*
Generators Have a `send` Method

- Generators have a `send()` method
  - `a = mygenerator()`
  - `b = next(a)  # progress and get a value`
  - `a.send(val)  # sends a value back`

- Sends to a `yield expression`
  - **Format**: `(yield)  # parentheses are necessary`
  - Typically used in an assignment
  - **Example**: `value = (yield)`
Generators Have a **send** Method

- Generators have a `send()` method
  - `a = mygenerator()`
  - `b = next(a)`  # progresses to next value
  - `a.send(val)`  # sends a value back

- Sends to a **yield expression**
  - **Format:** `(yield)`  # parentheses are necessary
  - Typically used in an assignment
  - **Example:** `value = (yield)`
Visualizing in the Tutor

```python
def receive(n):
    """Receives n values as input""
    for x in range(n):
        # receive the value
        value = (yield)
        print('Coroutine received value ' + repr(value))

    # Add this if using the Python Tutor
    a = receive(3)
    next(a)  # Get the thing started
    a.send('x')
    a.send('y')
```

**next() takes us to first yield**
Visualizing in the Tutor

```python
1 def receive(n):
2     '''Receives n values as input and prints them
3     for x in range(n):
4         # receive the value sent
5         value = (yield)
6     print('Coroutine received value ' + repr(va
7
8
9     # Add this if using the Python Tutor
10    a = receive(3)
11    next(a)  # Get the thing started
12    a.send('x')
13    a.send('y')
```

Resumes with a new variable!
Visualizing in the Tutor

```python
1 def receive(n):
2     """Receives n values as input and prints them
3         for x in range(n):
4             # receive the value sent
5             value = (yield)
6         print('Coroutine received value '+repr(value))
7
8 # Add this if using the Python Tutor
9 a = receive(3)
10 next(a)  # Get the thing started
11 a.send('x')
12 a.send('y')
```

Continue to move forward with send()
Can Do Both Output and Input

- **Format:** `var = (yield expr)`
  - Coroutine evaluates `expr` and outputs it
  - Coroutine stops and lets parent resume
  - When coroutine resumes, new value in `var`

- **Example:**

  ```python
  def give_receive(n):
      """Receives n values as input and prints them"""
      for x in range(n):
          value = (yield x)
          print('Received ' + repr(value))
  ```
Visualizing Back-and-Forth

def give_receive(n):
    
    # Give x to the parent function, receive
    # value = (yield x)
    print('Coroutine received value ' + repr(value))

# Add this if using the Python Tutor
a = give_receive(3)

x = next(a)  # Get the first value in yield p
y = a.send('x')  # Also returns the yield value i
z = a.send('y')

next() gets first value from yield
Visualizing Back-and-Forth

```python
def give_receive(n):
    """Receives n values as input and prints them
    for x in range(n):
        # Give x to the parent function, receive
        value = (yield x)
    print('Coroutine received value ' + repr(value))

    # Add this if using the Python Tutor
    a = give_receive(3)
    x = next(a)  # Get the first value in yield point
    y = a.send('x')  # Also returns the yield value i.e. 0
    z = a.send('y')
```

**send() makes new variable**
Visualizing Back-and-Forth

```python
1 def give.receive(n):
2     '''Receives n values as input and prints them
3     for x in range(n):
4         # Give x to the parent function, receive
5         value = (yield x)
6         print('Coroutine received value ' + repr(value))

7     # Add this if using the Python Tutor
8     a = give.receive(3)
9     x = next(a)  # Get the first value in yield p
10    y = a.send('x')  # Also returns the yield value i
11    z = a.send('y')
```

Program output:
yield outputs the expression
Visualizing Back-and-Forth

```python
def give_receive(n):
    """Receives n values as input and prints them"
    for x in range(n):
        # Give x to the parent function, receive value = (yield x)
        value = (yield x)
    print('Coroutine received value ' + repr(value))

# Add this if using the Python Tutor
a = give_receive(3)
x = next(a)  # Get the first value in yield p
y = a.send('x')  # Also returns the yield value i
z = a.send('y')
```

Program output:

```
return value of send()
```
Application: Animation Smoothing

- Our animation sequence is **timed**
  - We needed to keep track of the time
  - Did that with the constant `FRAME_RATE`
  - Assumes a consistent **60 frames per second**

- But what if we do not actually have that?
  - The animation will be jerky (**this is okay**)
  - The animation will run too long (**this is not**)

- **Example:** Set `MAKE_LAG` to `True`
def _animate_turn(self, direction):
    """Animates a rotation of the image over SPEED seconds"""
    sangle = self.image.angle
    fangle = sangle + 90 if direction == 'left' else sangle - 90
    steps = (fangle - sangle) / ANIMATION_SPEED  # Compute degrees per second
    animating = True
    while animating:
        dt = (yield)  # Get time to animate
        amount = steps * dt
        self.image.angle = self.image.angle + amount  # Update the angle
        if abs(self.image.angle - sangle) >= 90:
            self.image.angle = fangle
            animating = False
def _animate_turn(self, direction):
    """Animates a rotation of the image over SPEED seconds"""
    sangle = self.image.angle
    fangle = sangle + 90 if direction == 'left' else sangle - 90
    steps = (fangle - sangle) / ANIMATION_SPEED  # Compute degrees per second
    animating = True
    while animating:
        dt = (yield)  # Get time to animate
        amount = steps * dt
        self.image.angle = self.image.angle + amount  # Update the angle
        if abs(self.image.angle - sangle) >= 90:
            self.image.angle = fangle
            animating = False
            animating = False
def update(self, dt):
    
    """Animates the image."""
    if not self._animator is None:  # Something to animate
        try:
            self._animator.send(dt)  # Tell it secs to animate
        except StopIteration:
            self._animator = None  # Stop animating
    elif self.input.is_key_down('left'):
        self._animator = self._animate_turn('left')
        next(self._animator)  # Start up the animator
    ...
def update(self, dt):
    """Animates the image."""
    if not self._animator is None:
        try:
            self._animator.send(dt)
            if self.input.is_key_down('left'):
                self._animator = self._animate_turn('left')
                next(self._animator)
        except:
            self._animator = None
    # Something to animate
    # Stop animating
    # Start coroutine after creating it
    # Send dt to the yield expression
Coroutines and Animation

- Popular approach in Unity game engine
  - Coding is in C#, not Python
  - But it has a yield and coroutines
- Because the Unity engine is complicated
  - Will not let you touch the core loop
  - You can only add custom animation scripts
  - With coroutines, get to program with the loop
- This is all cutting edge!
  - C++ added coroutines in 2020
Optional Exercise
New Application: Counting Words

counts = {}  # Store the word count
word = ""  # Accumulator to build word
for x in text:
    if x.isalpha():  # Word continues
        word = word+x
    else:  # Word ends
        # Add it if not empty
        if word != ":
            add_word(word,counds)
        word = ""  # Reset the accumulator

What if text is really long?
Progress Monitoring

- Want some way to measure progress
  - Graphical progress bar
  - Or even just print statements
- But do not want it inside the function
  - Want the user to be able to customize this
  - So the calling function monitors progress
- No way to do with simple function
  - We only know the progress when complete
for pos in range(len(text)):
    if pos % interval == 0:
        yield progress
    if x.isalpha():  # Word continues
        word = word + x
    else:  # Word ends
        # Add it if not empty
        if word != "":
            add_word(word, counts)
        word = ""  # Reset the accumulator

Periodically notify caller
The Parent Caller

```
loader = wordcount(file)       # Create coroutine
result = None

# Keep going as long as the loader has more
while not loader is None:
    try:
        amount = next(loader)   # Load some more data
        show_progress(amount)
    except StopIteration as e:
        result = e.args[0]     # Access the return value
        loader = None          # We are done
```

read1.py