Module 29

Coroutines
What is Multitasking?

• In CS 1110 you create simple programs
  ▪ You run the script in the Terminal
  ▪ Program runs until done (or you quit)
  ▪ Only then does Terminal “return” control

• But computers multiple programs at once
  ▪ We can switch between without quitting
  ▪ Some run simultaneously (playing music)
  ▪ This is what we call **multitasking**

• Can we do something like this in Python?
## But There are Two Types

<table>
<thead>
<tr>
<th>Concurrency</th>
<th>Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>• All programs <em>make progress</em></td>
<td></td>
</tr>
<tr>
<td>▪ Switch between programs</td>
<td></td>
</tr>
<tr>
<td>▪ Switches are very fast (μs)</td>
<td></td>
</tr>
<tr>
<td>• Looks/feels simultaneous</td>
<td></td>
</tr>
<tr>
<td>• Programs <em>run at same time</em></td>
<td></td>
</tr>
<tr>
<td>▪ Each program gets CPU/core</td>
<td></td>
</tr>
<tr>
<td>▪ No switching between progs</td>
<td></td>
</tr>
<tr>
<td>• Actually is simultaneous</td>
<td></td>
</tr>
</tbody>
</table>

- **Multitasking on old hardware**
- **Multitasking on modern hardware**
An Important Distinction

Concurrency

Parallelism
## Switching in Currency

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

## 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 **threads**

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 again relevant to graphical applications
  ▪ They make a lot of animation code easier
  ▪ Used heavily by the Unity game engine
Terminology: 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><strong>Subroutine</strong></th>
<th><strong>Coroutine</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>• Runs until completed</td>
<td>• Can stop and start</td>
</tr>
<tr>
<td>▪ Invoked by parent routine</td>
<td>▪ Runs for a little while</td>
</tr>
<tr>
<td>▪ Runs until reach the end</td>
<td>▪ Returns control to parent</td>
</tr>
<tr>
<td>▪ Returns output to parent</td>
<td>▪ And then picks up again</td>
</tr>
<tr>
<td>• Just like a function call</td>
<td>• <em>Kind of</em> like a generator</td>
</tr>
<tr>
<td>▪ Parent is “frozen”</td>
<td>▪ Starts up at initial call</td>
</tr>
<tr>
<td>▪ Subroutine/function runs</td>
<td>▪ Can yield execution</td>
</tr>
<tr>
<td>▪ Parent resumes when done</td>
<td>▪ Resumes with full state</td>
</tr>
</tbody>
</table>
Subroutines vs Coroutines

Subroutine

Program 1

Frozen

Program 2

Coroutine

Program 1

Frozen

Program 2

Frozen
Subroutines vs Coroutines

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Coroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program 1</td>
<td>Program 1</td>
</tr>
<tr>
<td>Program 2</td>
<td>Program 2</td>
</tr>
</tbody>
</table>

- **Subroutine**
  - Frozen
  - call
  - return

- **Coroutine**
  - Frozen
  - yield
  - next
  - Frozen
  - yield
  - next
  - Frozen
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,counts)
        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

```python
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
Can Interleave Multiple Coroutines

```
loader1 = wordcount(file1)
loader2 = wordcount(file2)

progress1 = next(loader1)
progress2 = next(loader2)
progress1 = next(loader1)
progress2 = next(loader2)
progress2 = next(loader2)
```

....

read2.py
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
  ▪ There
  ▪ Yield
  ▪ But

Need another command
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 and gets 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)`
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      
3          
4          
5  # receive the value sent
6  value = (yield)
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))
  ```
def give_receive(n):
    
    for x in range(n):
        # Give x to the parent function, receive value = (yield x)
        value = (yield x)
        print('Coroutine received value ' + repr(va)
    
    # 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

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(x))
    # 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')

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

```python
1 def give_receive(n):
2     
3         for x in range(n):
4             # Give x to the parent function, receive
5             value = (yield x)
6             print('Coroutine received value ' + repr(va
7
8             # Add this if using the Python Tutor
9             a = give_receive(3)
10            x = next(a)     # Get the first value in yield p
11            y = a.send('x') # Also returns the yield value i
12            z = a.send('y')
```

Program output:

```
yield outputs the expression
```
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
    z = a.send('y')

return value of send()
Animation in Assignment 7

• **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
def update(self, dt):
    """Animates the image."""
    if self._animating:
        if self._rotation:
            self._animate_turn(dt)
        else:
            self._animate_slide(dt)
    elif self.input.is_key_down('left'):
        self._animating = True
        self._rotation = True
        self._sangle = self.image.angle
        self._fangle = self._sangle + 90
    ...

Ignore input if still animating

Otherwise start animation for given input
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*dt
    # 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, dt):
    """Animates a rotation of the image over SPEED seconds""
    # Compute degrees per second
    steps = (self._fangle - self._sangle) / SPEED
    amount = steps * dt
    # 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, dt):
    # Compute degrees per second
    steps = (self._fangle - self._sangle) / SPEED
    amount = steps * dt
    # 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

Is there a simpler way to do this?
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:
            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:  # Something to animate
        try:
            self._animator.send(dt)  # Tell it secs to animate
        except:
            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
        ...  # Ignore input if still animating
        ...  # Otherwise start animation for given input
**Same Animation with Coroutines**

```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:
        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)
        amount = steps * dt
        self.image.angle = self.image.angle + amount
        if abs(self.image.angle - sangle) >= 90:
            self.image.angle = fangle
            animating = False

    Loop is explicit. Animate until done
Another Application: Time Budgeting

- Update
- Draw

Function calls
Another Application: Time Budgeting

60 fps!

Function

Must finish in 0.016 seconds

calls

Update

Draw
With a Coroutine

60 fps!

You have 0.016 seconds

This is how loading screens work!
Application: Counting Words

budget = (yield)  # Get the initial budget
start = time.time()  # Start the timer

for pos in range(len(text)):
    end = time.time()  # See if we have taken too long
    if end-start > budget:
        progress = round(100*pos/len(text))  # Notify progress, get new budget
        budget = (yield progress)  # Notify progress, get new budget
        start = time.time()  # Reset the timer for new budget

# Build up the word, one letter at a time
Application: Counting Words

budget = (yield)            # Get the initial budget
start = time.time()       # Start the timer
for pos in range(len(text)):
    end = time.time()     # See if we have taken too long
    if end-start > budget:
        progress = round(100*pos/len(text))
        budget = (yield progress)    # Notify progress, get new budget
        start = time.time()              # Reset the timer for new budget

# Build up the word, one letter at a time
Python Now Has Native Coroutines

- No longer just a generator variation
  - Supported since Python 3.5
  - Requires the asyncio module

- **Advantages**
  - A lot less code to write
  - Much is done for you automatically

- **Disadvantages**
  - Much less flexible
  - Cannot use it for animations in A7
Three Requirements

• The yield expression is replaced by `await`
  ▪ Stops until it gets an answer
  ▪ But it is not a yield; does not output anything

• Function/method must start with `async`
  ▪ Tells Python this is native coroutine
  ▪ Presence of `await` is not enough

• Must use `asyncio` to run the coroutines
  ▪ Top level function is `asyncio.run(…)`
  ▪ All helpers are `asyncio.create_task(…)`
```python
async def loadfiles(fname1,fname2):
    """Creates a word-count dictionary for fname1, fname2""
    # Create the tasks for the coroutines
    result1 = {}
    loader1 = asyncio.create_task(wordcount(fname1,result1))
    result2 = {}
    loader2 = asyncio.create_task(wordcount(fname2,result2))
    # Let them take over
    await loader1
    await loader2
    result = merge(result1,result2)
    print('Read a total of ' + str(len(result)) + ' words.')
```

The send() loop is handled for you
async def loadfiles(fname1,fname2):
    """Creates a word-count dictionary for fname1, fname2"""
    # Create the tasks for the coroutines
    result1 = {}
    loader1 = asyncio.create_task(wordcount(fname1,result1))
    result2 = {}
    loader2 = asyncio.create_task(wordcount(fname2,result2))
    # Let them take over
    await loader1
    await loader2
    result = merge(result1,result2)
    print('Read a total of '+str(len(result))+' words.')
Why Native Coroutines?

• The generator version is better!
  ▪ We have much more control
  ▪ The yield expression goes back-and-forth

• But native coroutines support **parallelism**
  ▪ Each coroutine can get its own core/thread
  ▪ The `await` command is how we synchronize

• It is possible to *emulate* a yield
  ▪ Requires very advanced Python (`pipe.py`)
  ▪ Beyond the scope of this course