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

Concurrency

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

Multitasking on **old** hardware

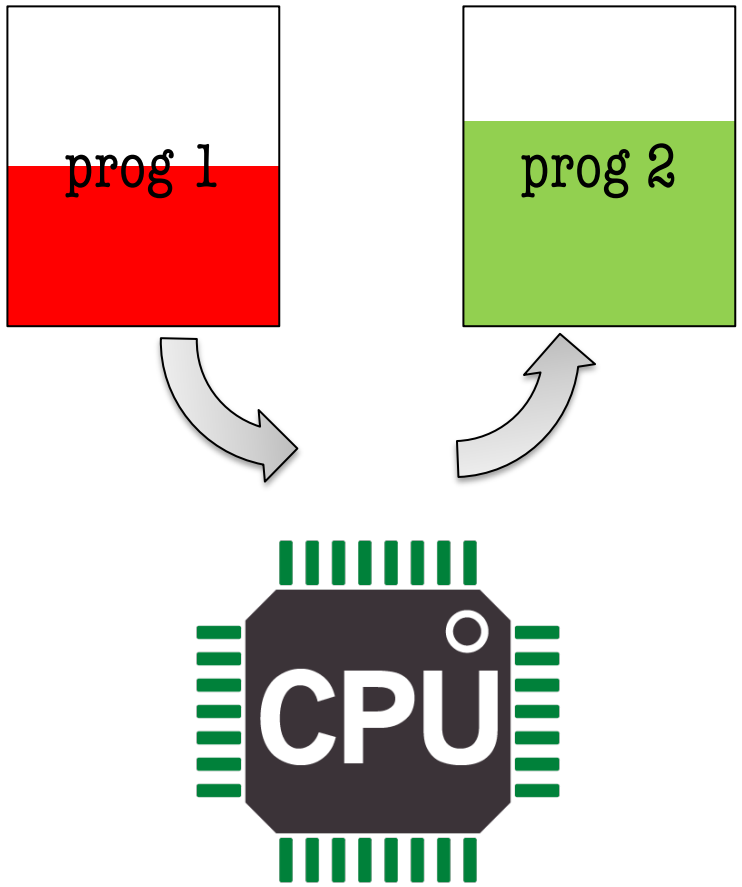
Parallelism

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

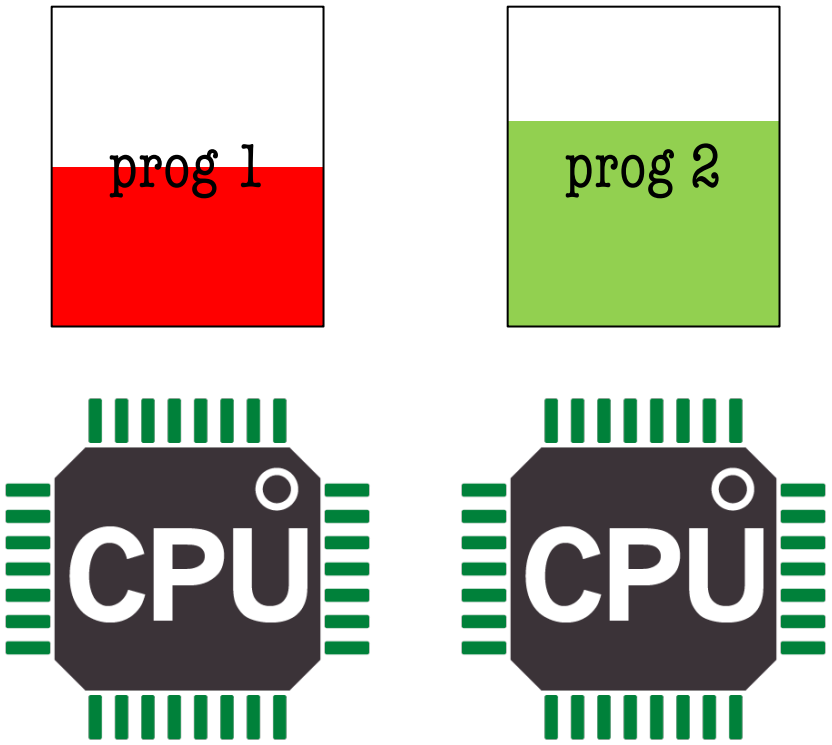
Multitasking on modern hardware

An Important Distinction

Concurrency



Parallelism



Switching in Currency

Preemptive

- 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

Cooperative

- 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
- Implement with **threads**
- 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
- Implement with **coroutines**
- Popular in early days of GUIs
  - Okay for main app to hog
  - No expectation of other apps

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

Subroutine

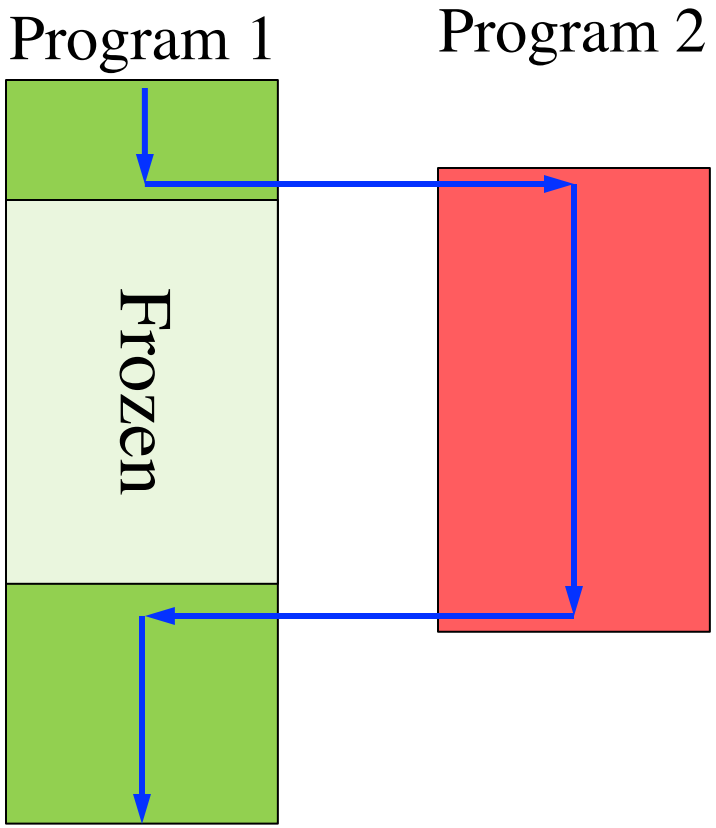
- Runs until completed
  - Invoked by parent routine
  - Runs until reach the end
  - Returns output to parent
- Just like a function call
  - Parent is “frozen”
  - Subroutine/function runs
  - Parent resumes when done

Coroutine

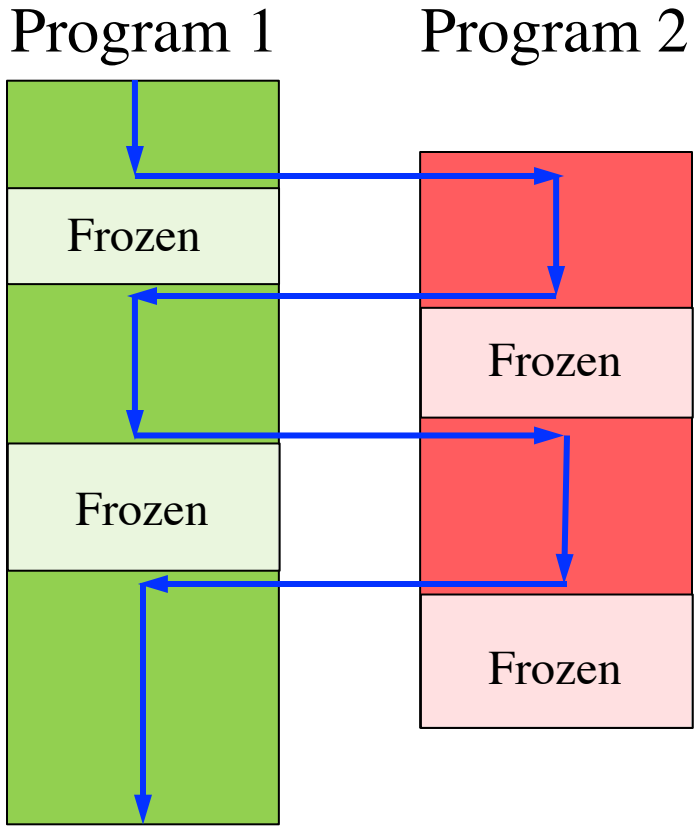
- Can stop and start
  - Runs for a little while
  - Returns control to parent
  - And then picks up again
- *Kind of* like a generator
  - Starts up at initial call
  - Can yield execution
  - Resumes with full state

Subroutines vs Coroutines

Subroutine

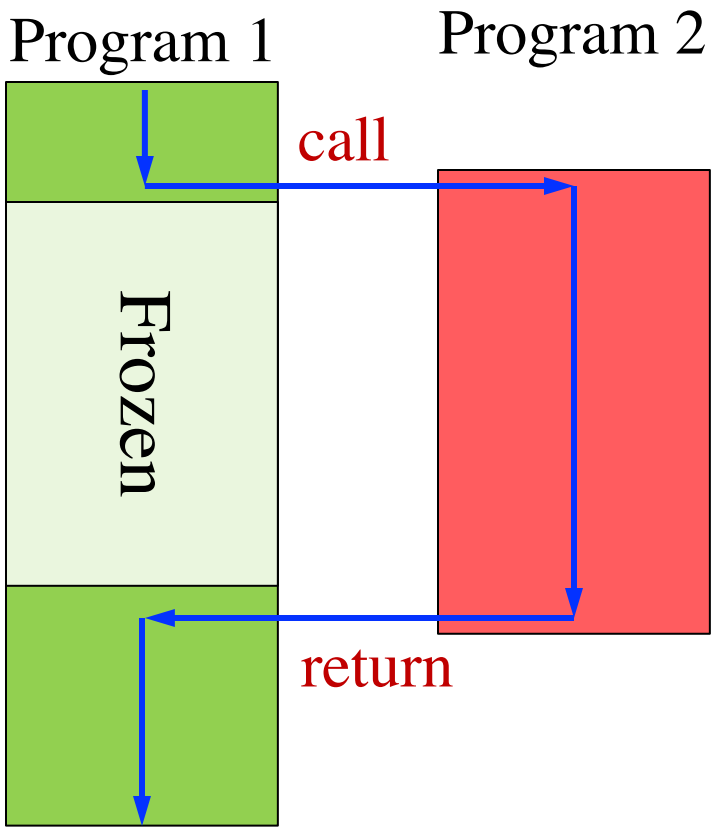


Coroutine

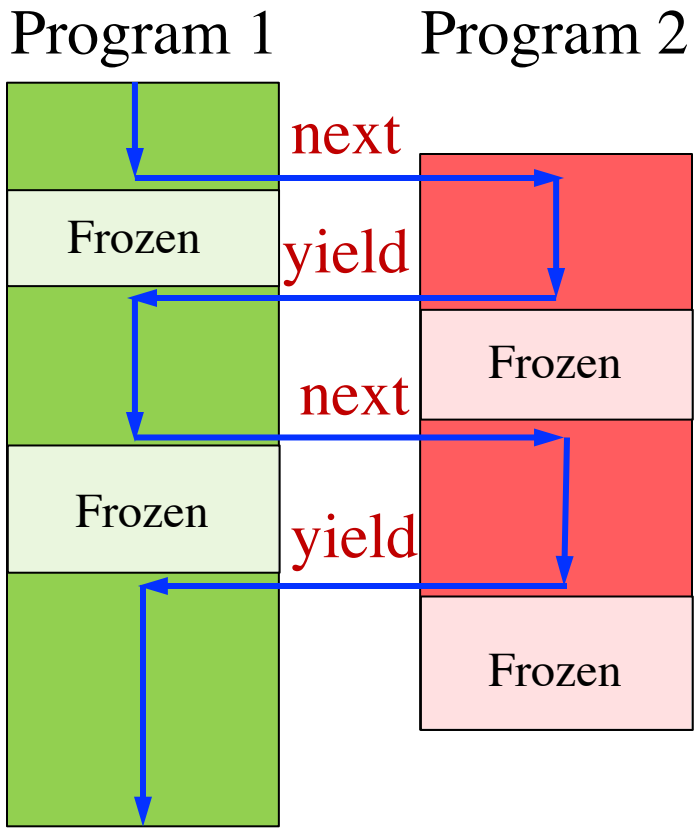


Subroutines vs Coroutines

Subroutine



Coroutine



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,counts)
        word = ''       # Reset the accumulator
```

read0.py

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

Application: Counting Words

```
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

readl.py

The Parent Caller

readl.py

```
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
```

Can Interleave Multiple Coroutines

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


progress1 = next(loader1)
progress2 = next(loader2)
progress1 = next(loader1)
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
  - The
  - Yie
  - 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
  - Pre
  - Fur

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)

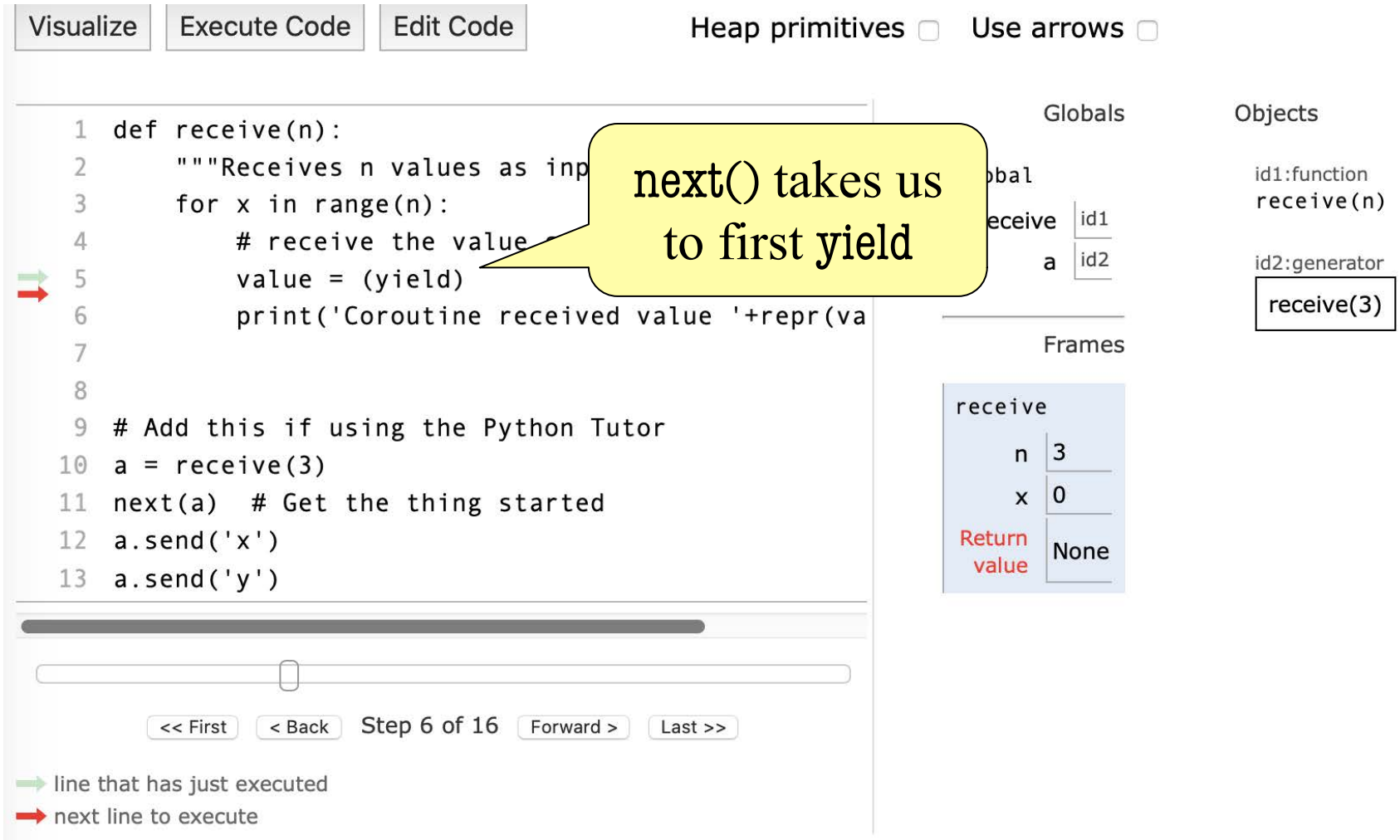
Generators Have a `send` Method

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

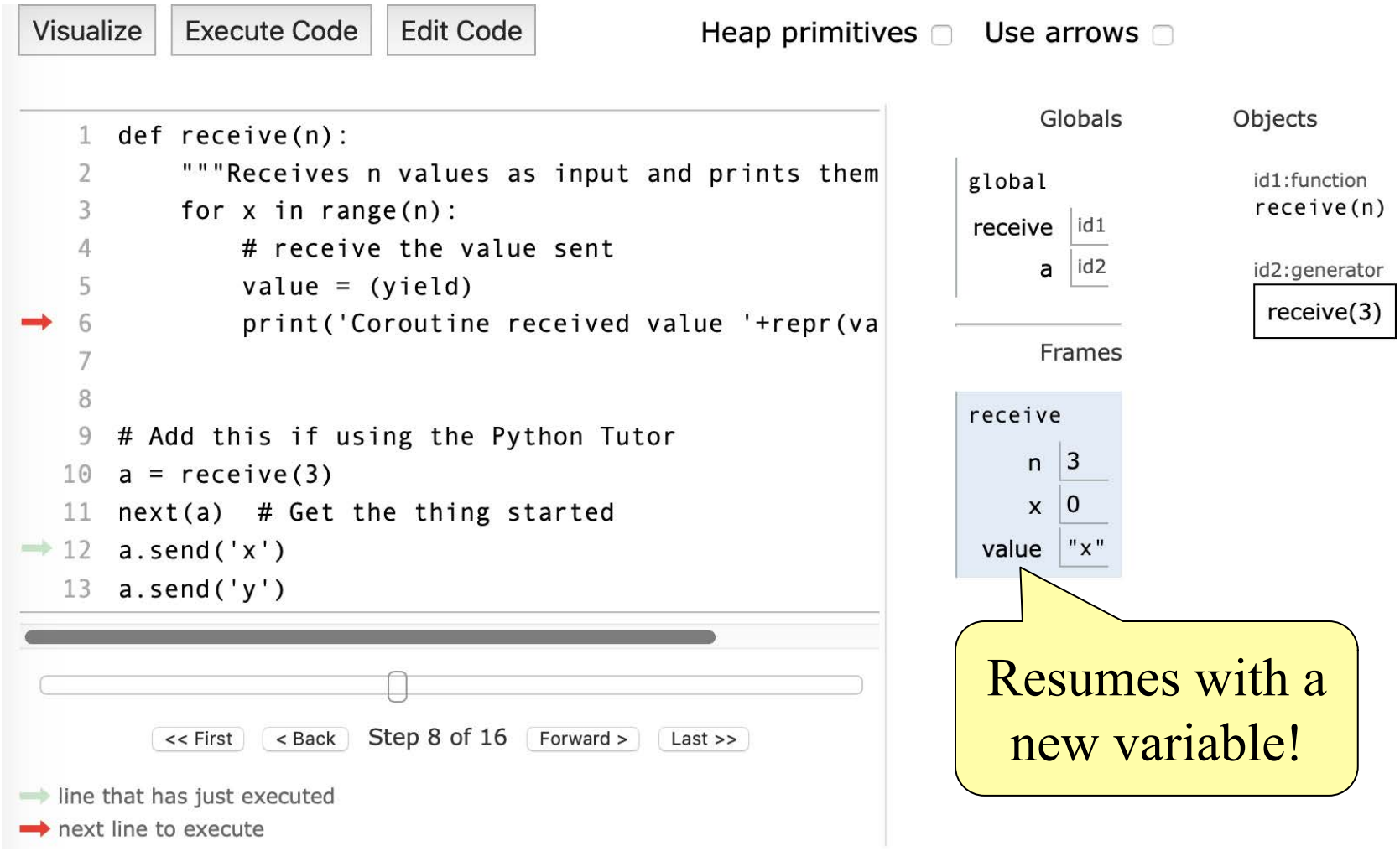
Must always start with next()

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

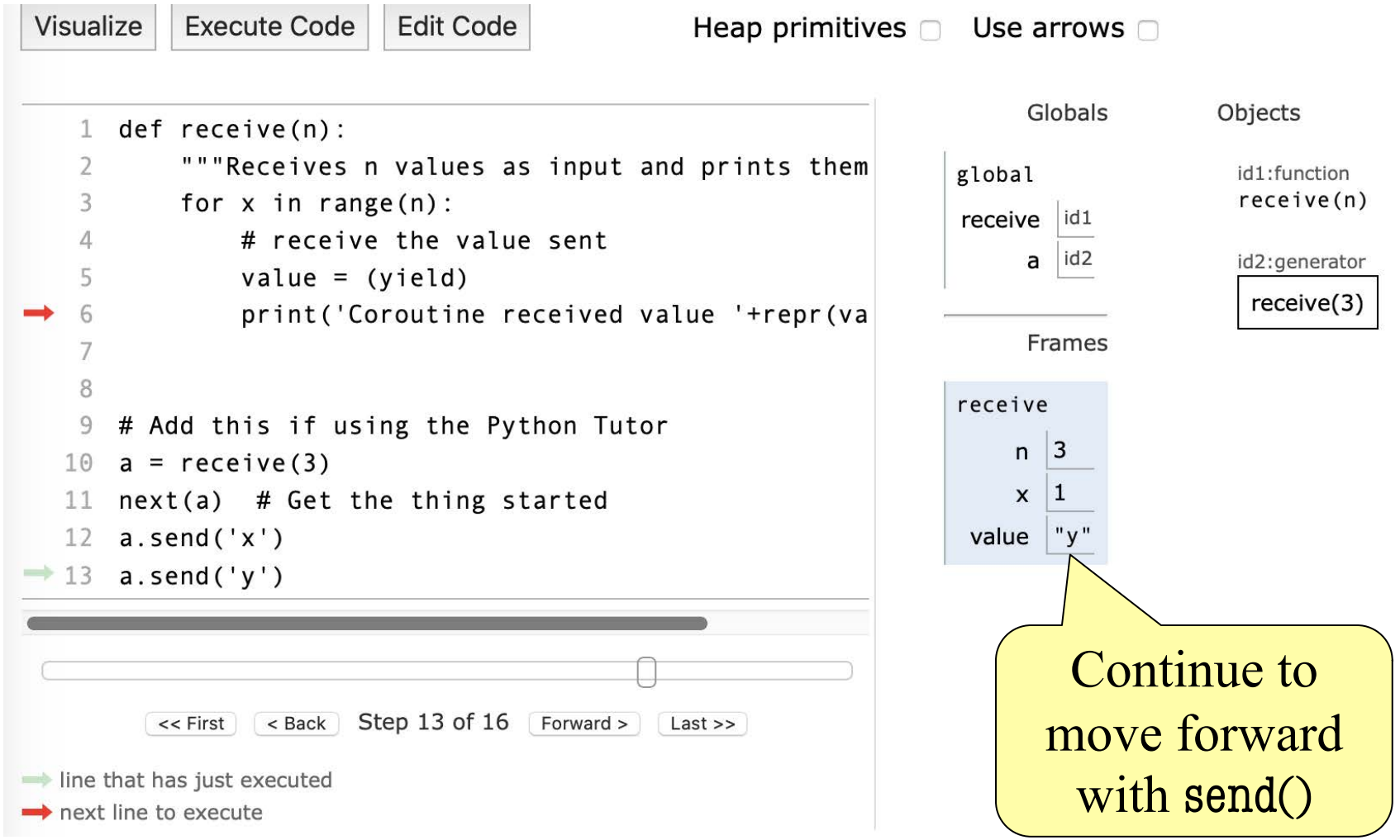
Visualizing in the Tutor



Visualizing in the Tutor



Visualizing in the Tutor

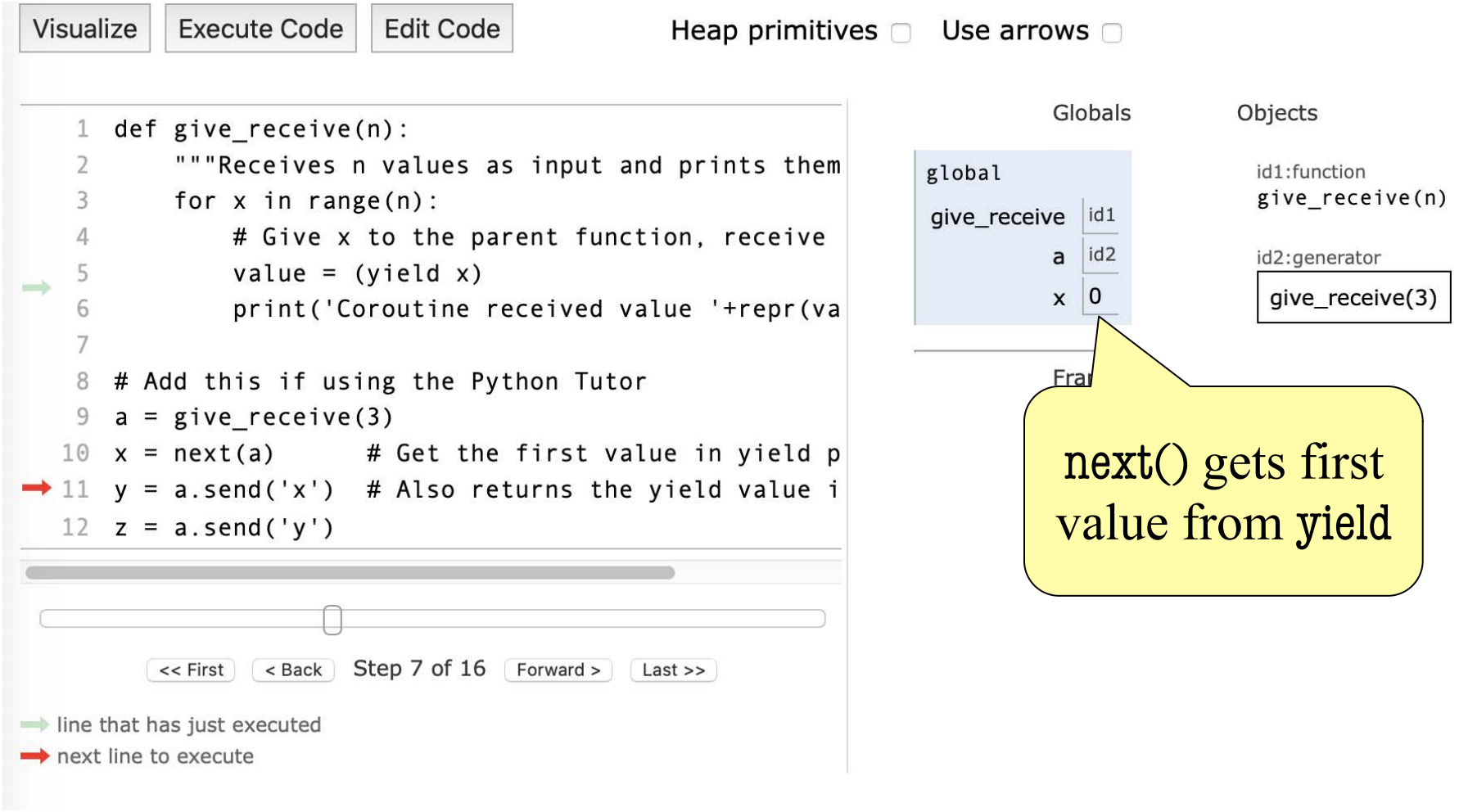


Can Do Both Ouput and Input

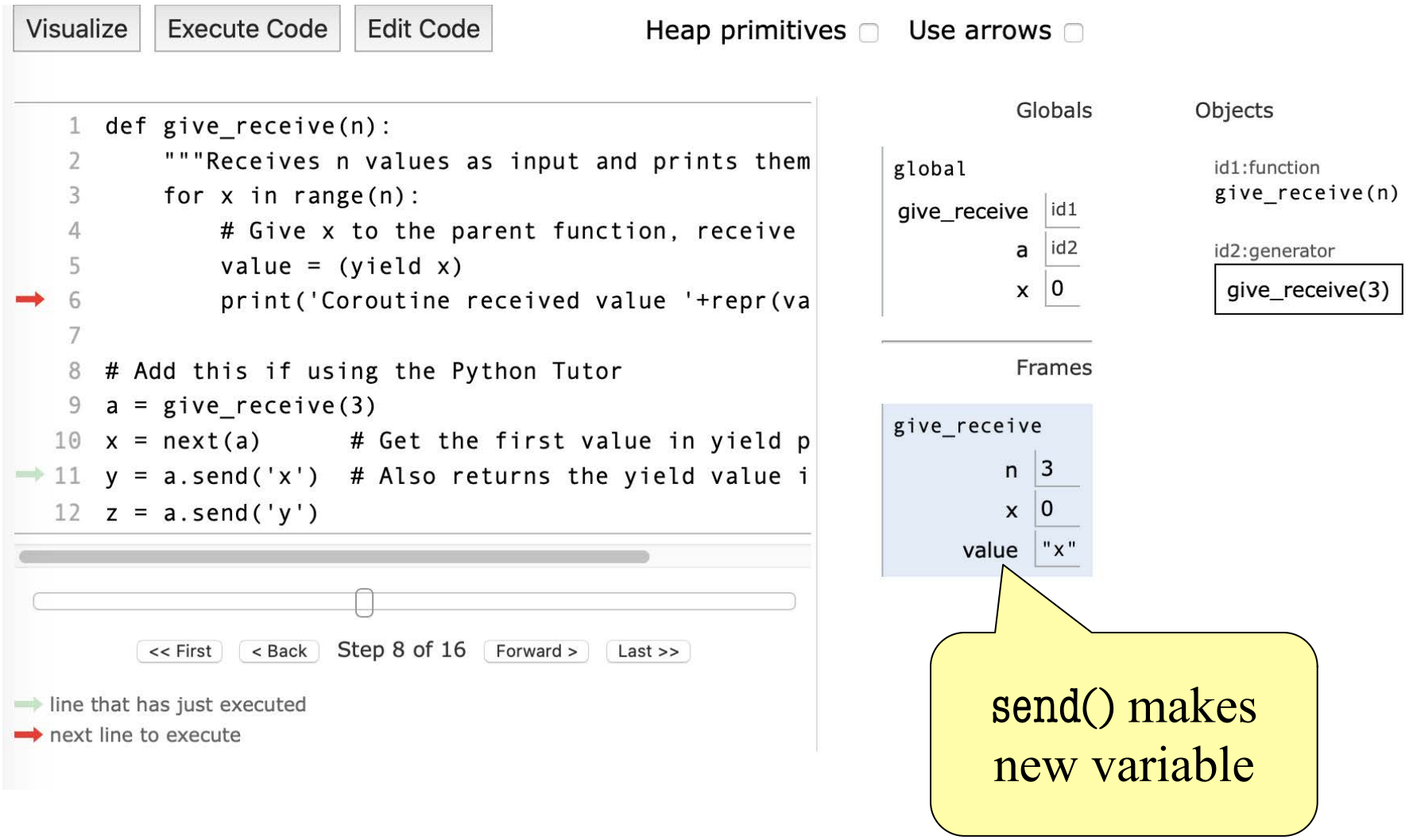
- **Format:** var = (yield expr)
  - Coroutine evaluates expr and outputs it
  - Coroutine stops and lets parent resume
  - When coroutine resumes, new value in var
- **Example:**

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

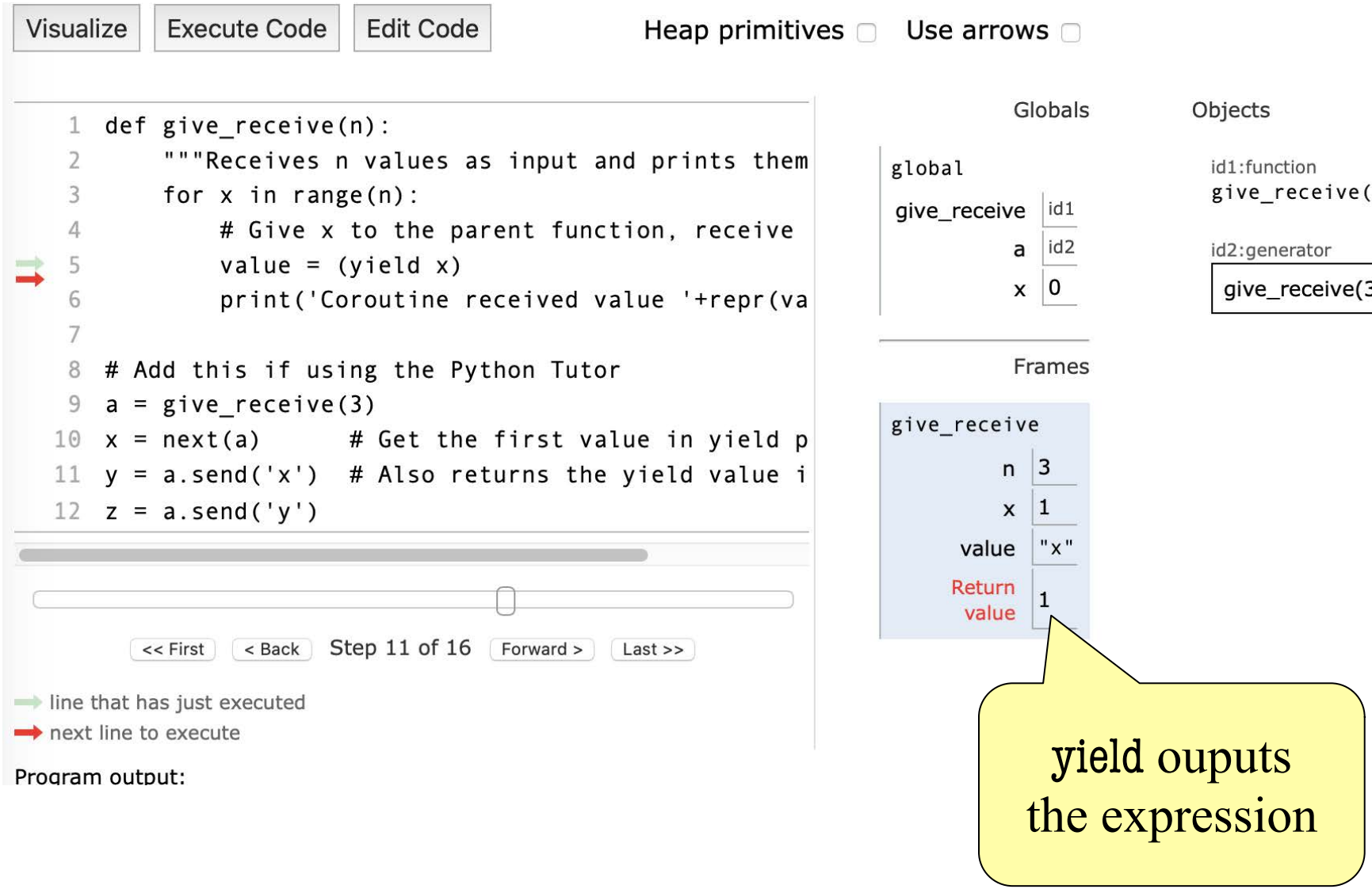
Visualizing Back-and-Forth



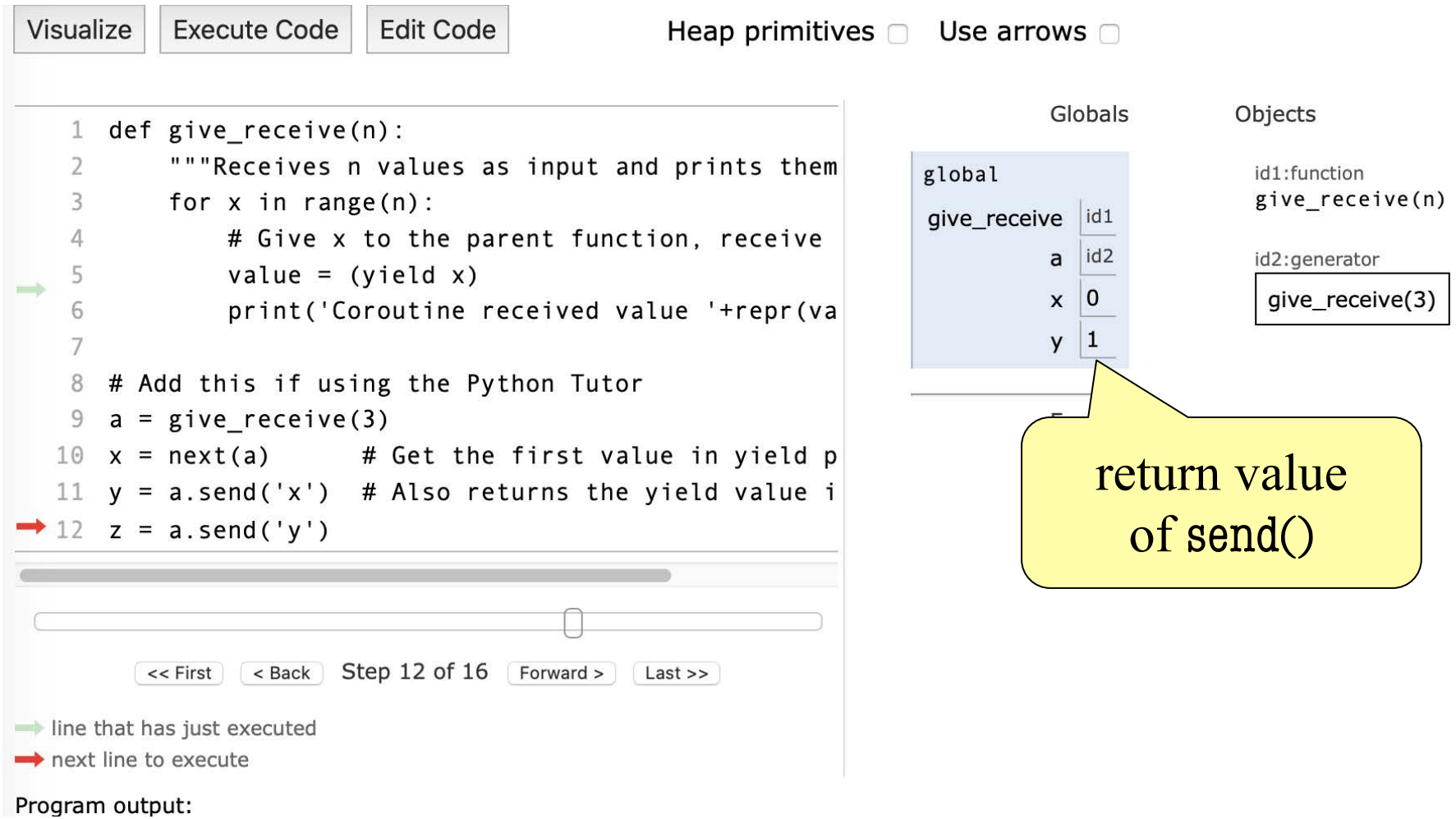
Visualizing Back-and-Forth



Visualizing Back-and-Forth



Visualizing Back-and-Forth



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

animate1.py

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

animate2.py

Timed Animation Example

```
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

Timed Animation Example

```
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 to far, clamp and stop animating
    if abs(self.image.angle-self._sangle) >= 90:
        self.image.angle = self._fangle
        self._animating = False
```

dt tells how far to animate

Animation Needs Many Attributes

```
def _animate_turn(self,dt):
    """n of the image over SPEED seconds"""
    # Compute deg s 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
    gle-self._sangle) >= 90:
        self.image. le = self._fangle
        self._animating = False
```

New Attribute

New Attribute

New Attribute

Animation Needs Many Attributes

```
def _animate_turn(self,dt):
    """n of the image over SPEED seconds"""
    # Compute de  s per second
    steps = (
    amount
    # Updat
    self.imag
    # If we go to far, clamp and stop animating
    gle-self._sangle) >= 90:
        self.image.  le = self._fangle
        self._animating = False
```

New Attribute

New Attribute

Is there a simpler way to do this?

Same Animation With Coroutines

```
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
            ...
```

Same Animation With Coroutines

```
def update(self,dt):
    """Animates the image."""
    if not self._animator is None:        # Something to animate
        try:
            self._animator.send(dt)       # Continue animating
        except:
            self._animator = None         # Stop animating
    elif self.input.is_key_down('left'):
        self._animator = self._animate_turn(-1)
        next(self._animator)              # Start animating
        ...
```

Ignore input if still animating

Otherwise start animation for given input

Same Animation with Coroutines

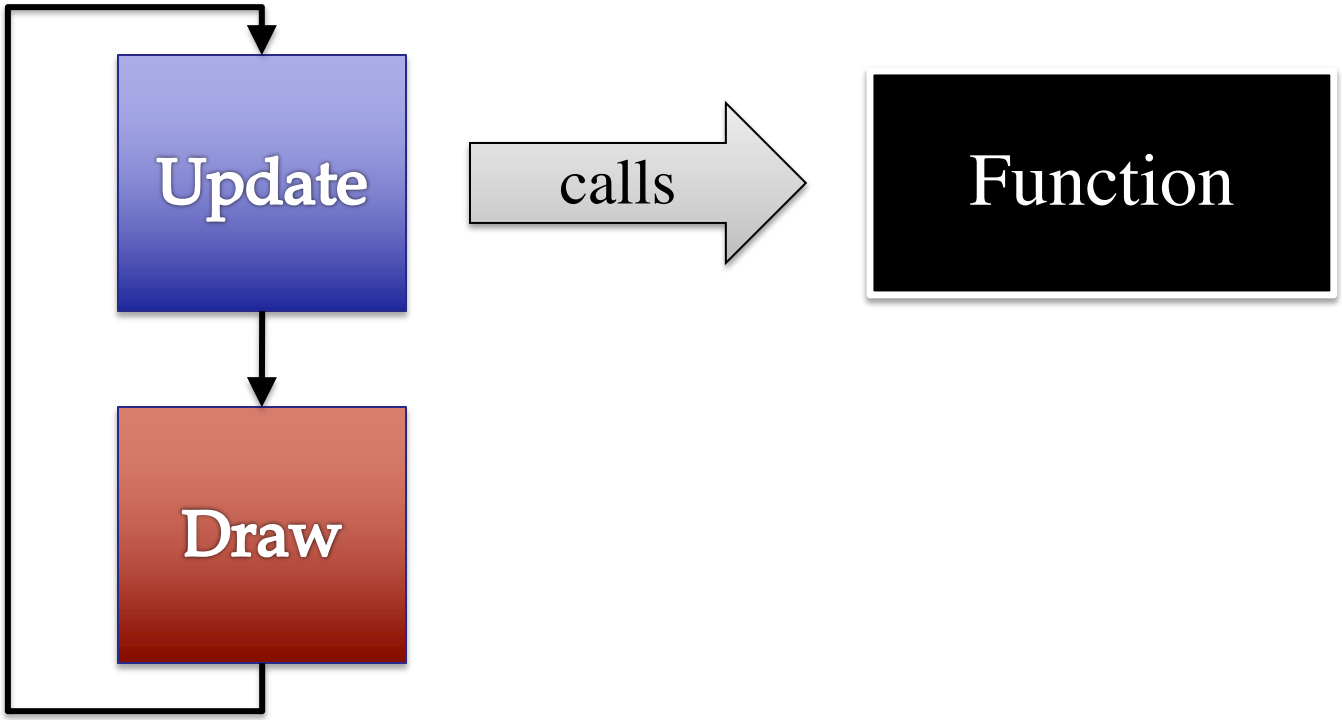
```
def _animate_turn(self,direction):
    """Animates a rotation of the image over SPEED seconds"""
    sangle = self.image.angle
    fangle = sangle+90 if direction == 'left' else sangle-90
    steps = (fangle-sangle)/ANIMATION_SPEED        # Compute degrees per second
    animating = True
    while 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
```

Same Animation with Coroutines

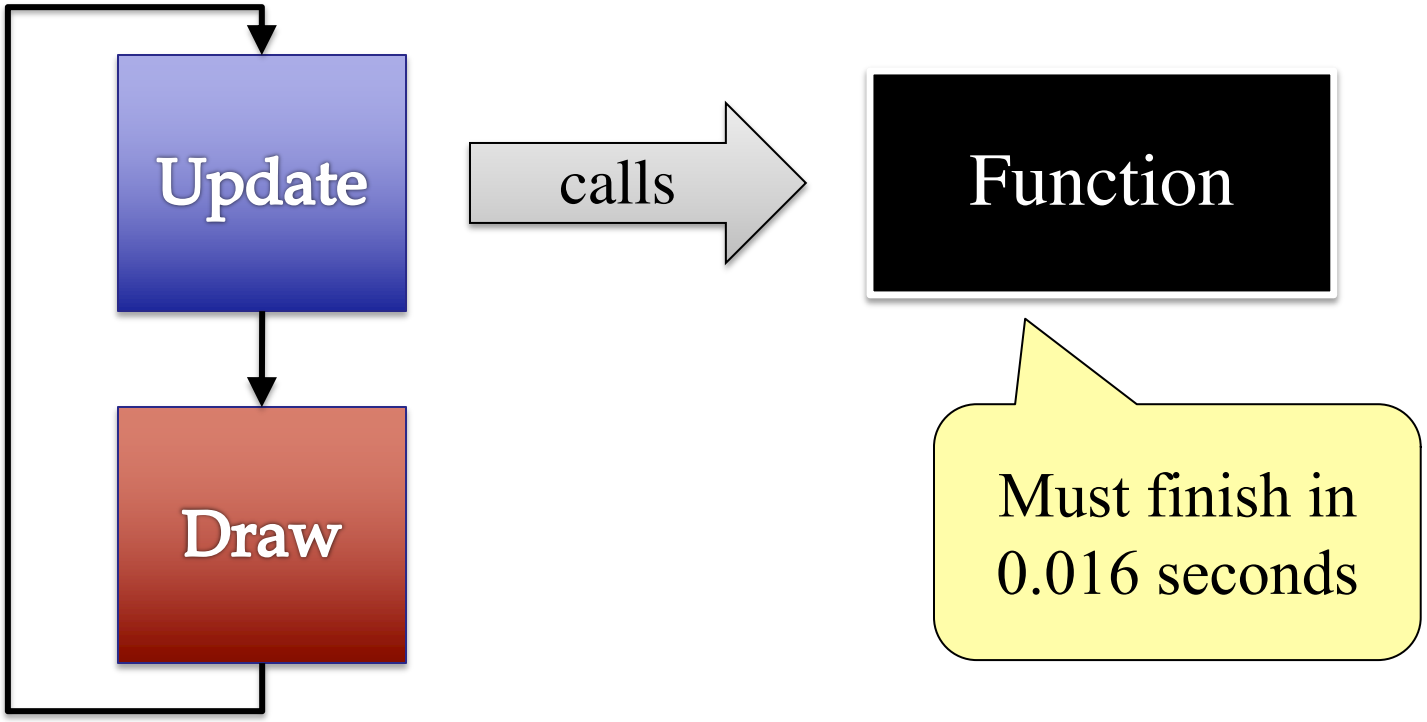
```
def _animate_turn(self,direction):
    """Animates a rotation of the image over SPEED seconds"""
    sangle = self.image.angle
    fangle = sangle+90 if direction == 'left' else sangle-90
    steps = (fangle-sangle)/ANIMATION_SPEED        # Compute degrees per second
    animating = True
    while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

Loop is explicit.
Animate until done

Another Application: Time Budgeting

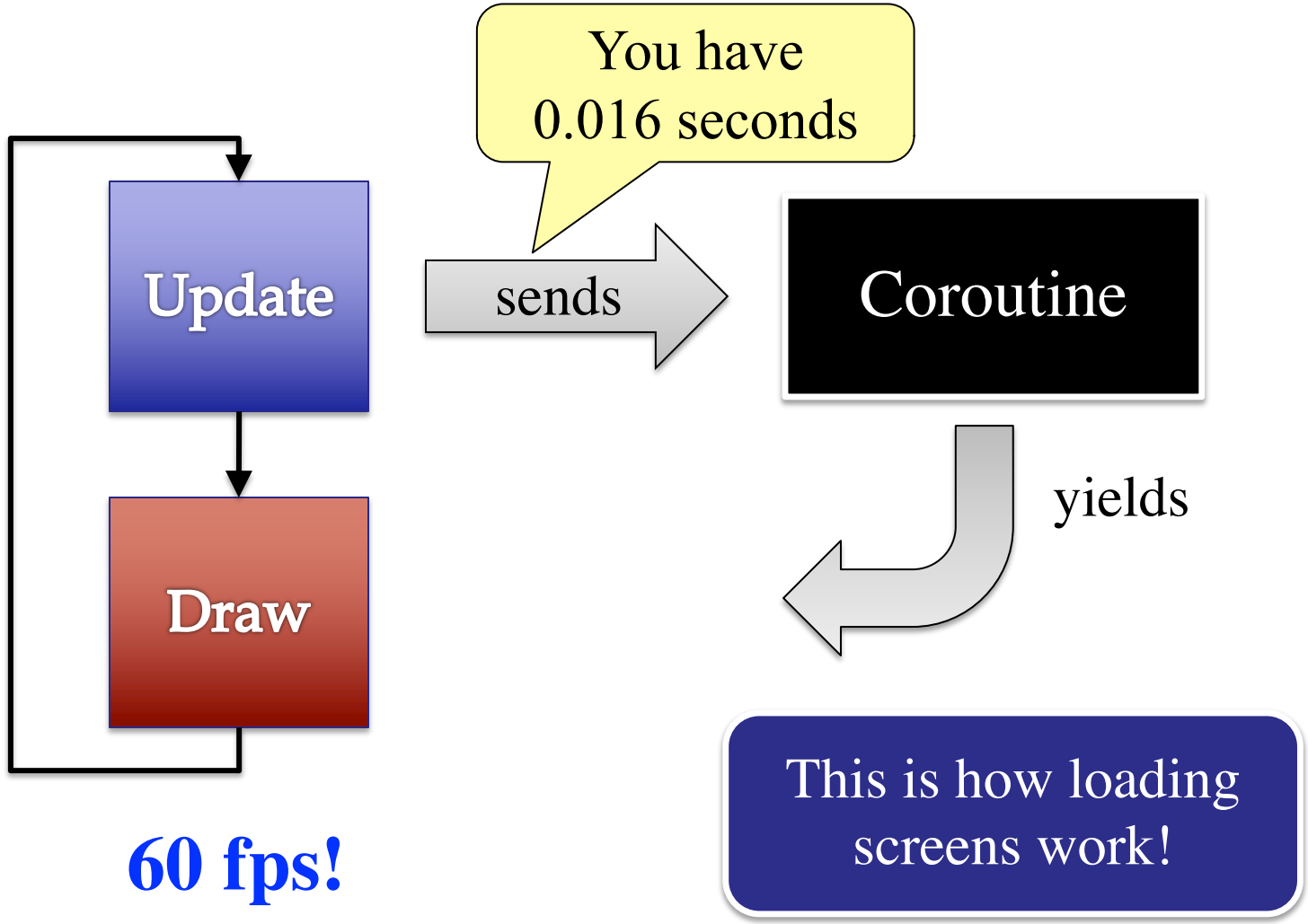


Another Application: Time Budgeting



60 fps!

With a Coroutine



Application: Counting Words

read3.py

```
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
```

Application: Counting Words

```
budget = (yield)                #                    get
start = time.time()             #
```

Current time in seconds

read3.py

```
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(...)`

Word Count Example Revisited

read4.py

```
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

Word Count Example Revisited

```
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.')
```

read4.py

But cannot get progress so far!

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