Lecture 25

Coroutines

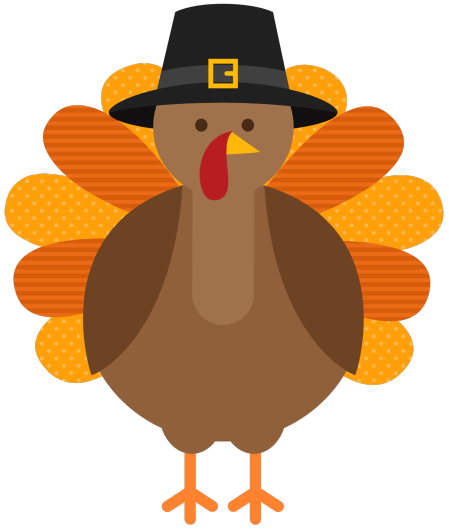
Announcements for This Lecture

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

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

Animation Needs Many Attributes

```
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
```

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*FRAME_RA
    # 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

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
```

Wouldn't a Loop Be Simpler?

```
def _animate_turn(
    """Animates a rotat      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-
        if abs(self.image.angle-sangle) >= 90:
            self.image.angle = fangle
            animating = False
```

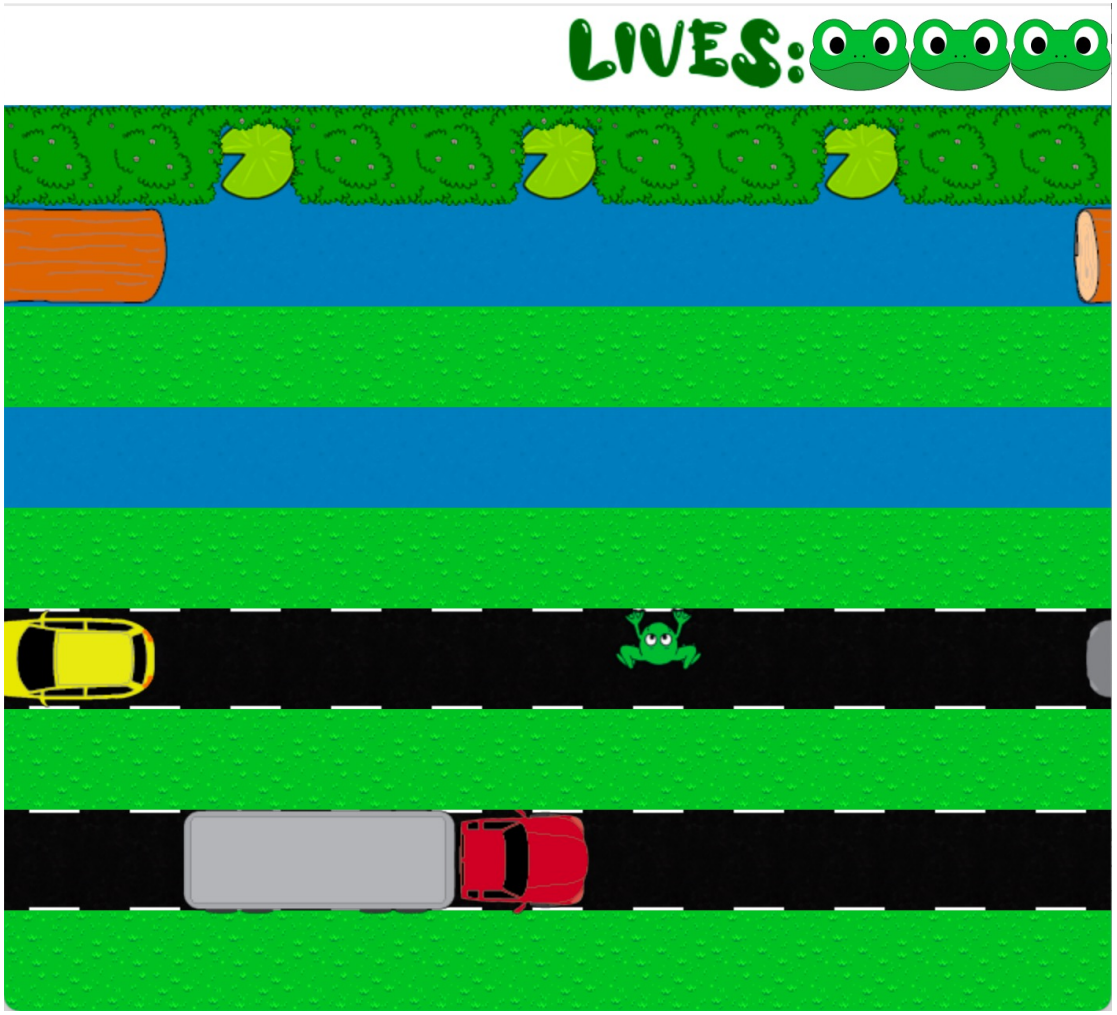
Only Attribute

Loop is explicit.
Animate until done.

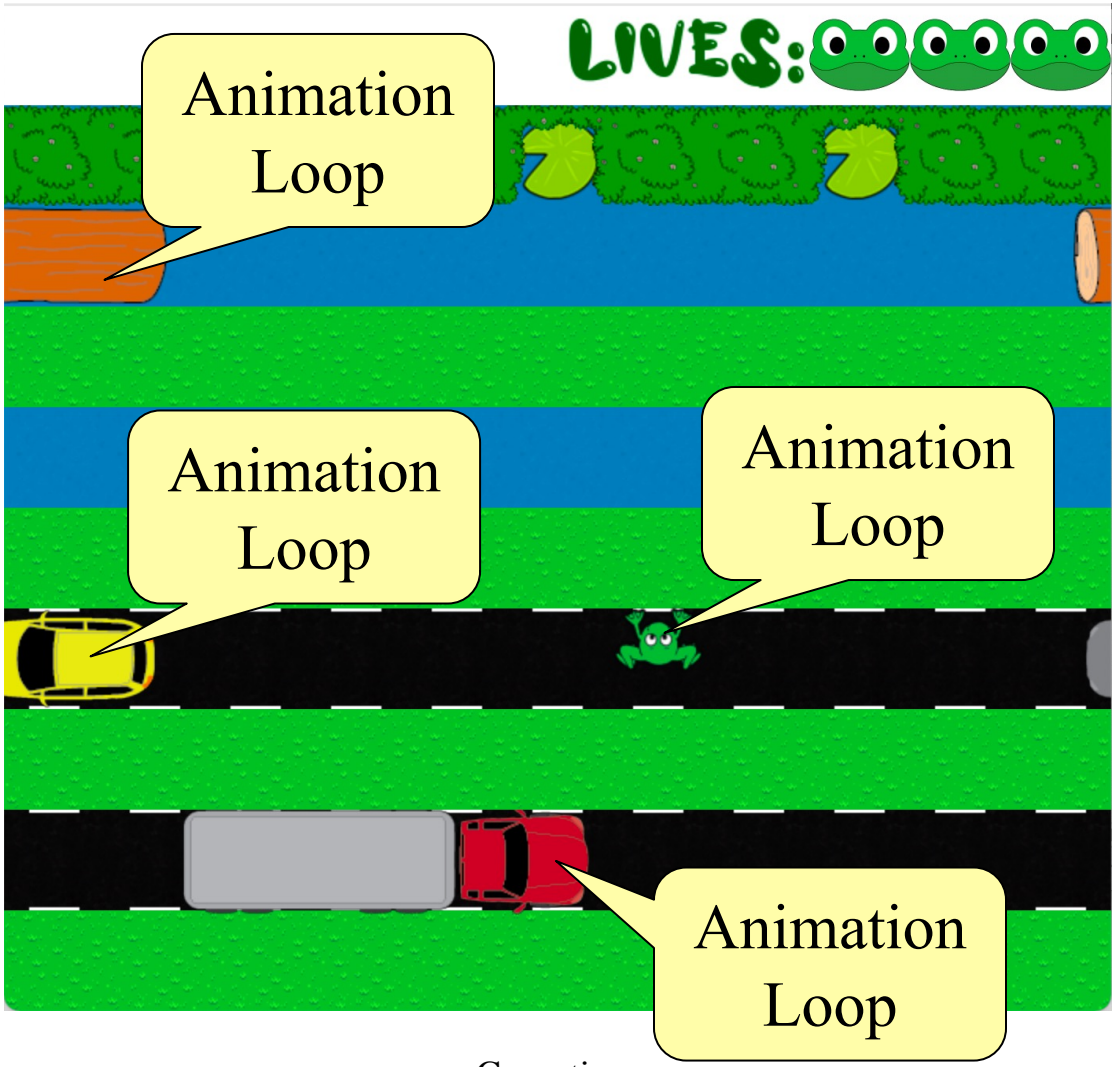
But This is Not Going to Work

- **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

What Do We Mean by Multitasking?

Concurrency

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

Multitasking on **old** hardware

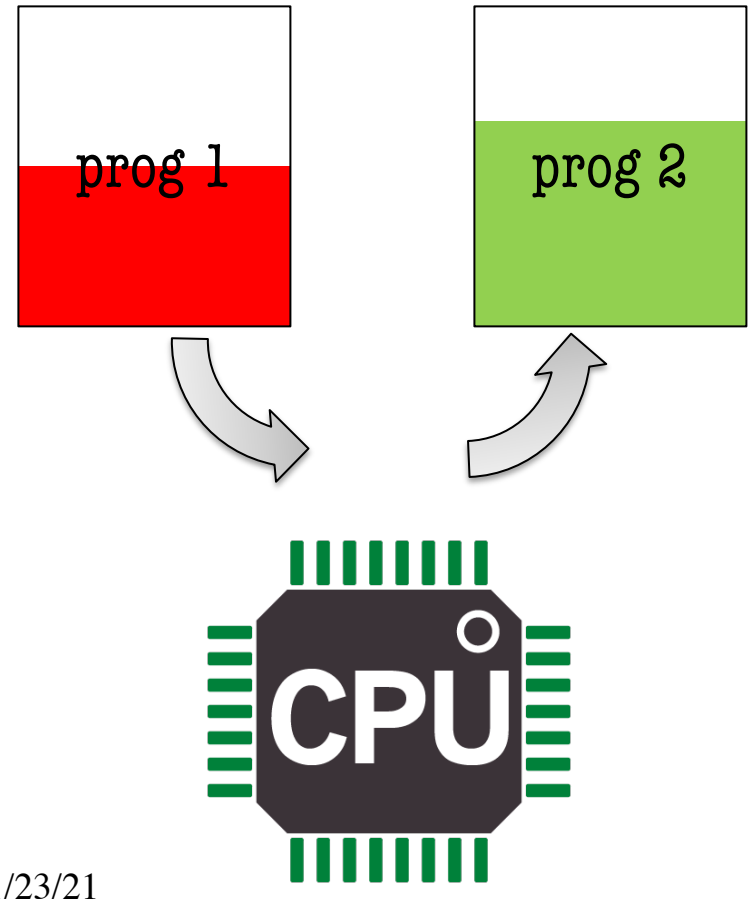
Parallelism

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

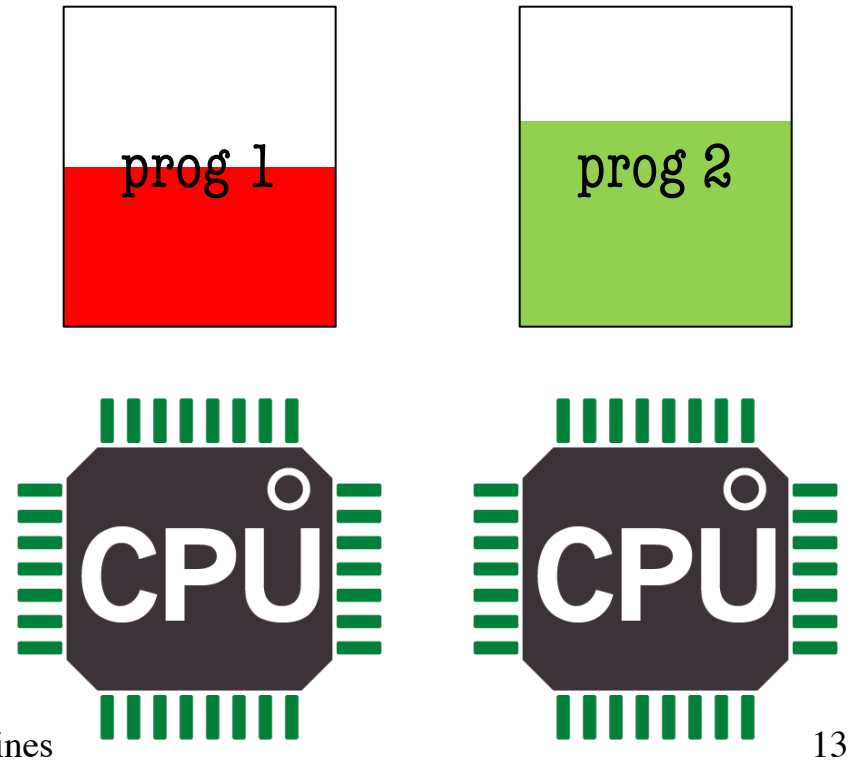
Multitasking on modern hardware

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 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

Subroutine

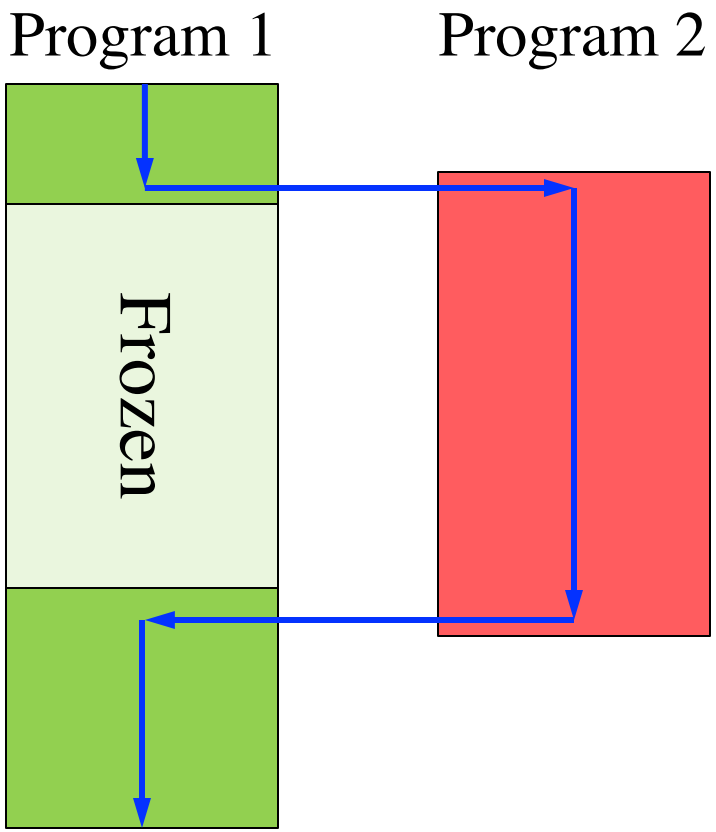
- Runs until completed
  - Invoked by parent routine
  - Runs until reach the end
  - Returns output to parent
- Just like a function call
  - Parent is “frozen”
  - Subroutine/function runs
  - Parent resumes when done

Coroutine

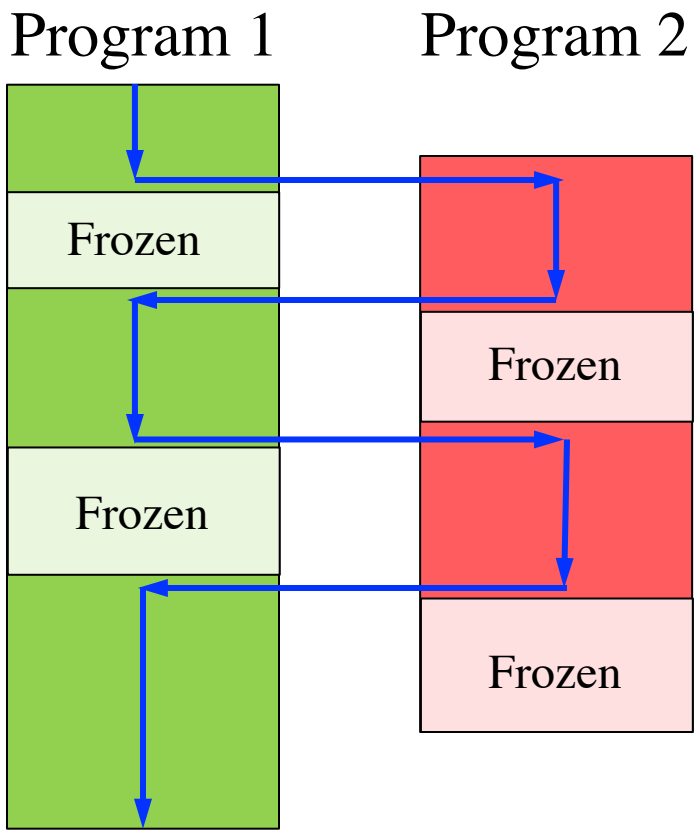
- Can stop and start
  - Runs for a little while
  - Returns control to parent
  - And then picks up again
- *Kind of* like a generator
  - Starts up at initial call
  - Can yield execution
  - Resumes with full state

Subroutines vs Coroutines

Subroutine

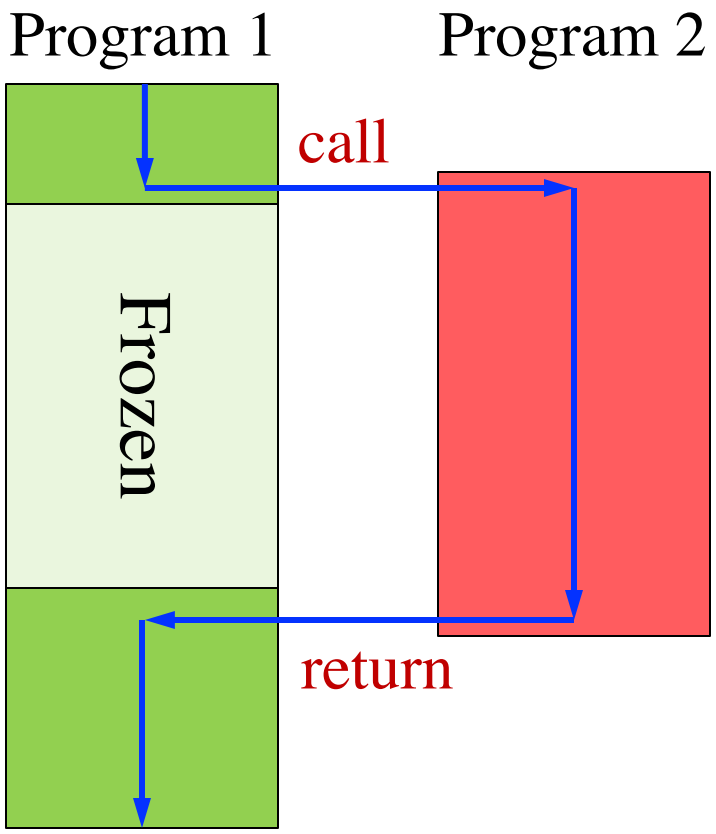


Coroutine

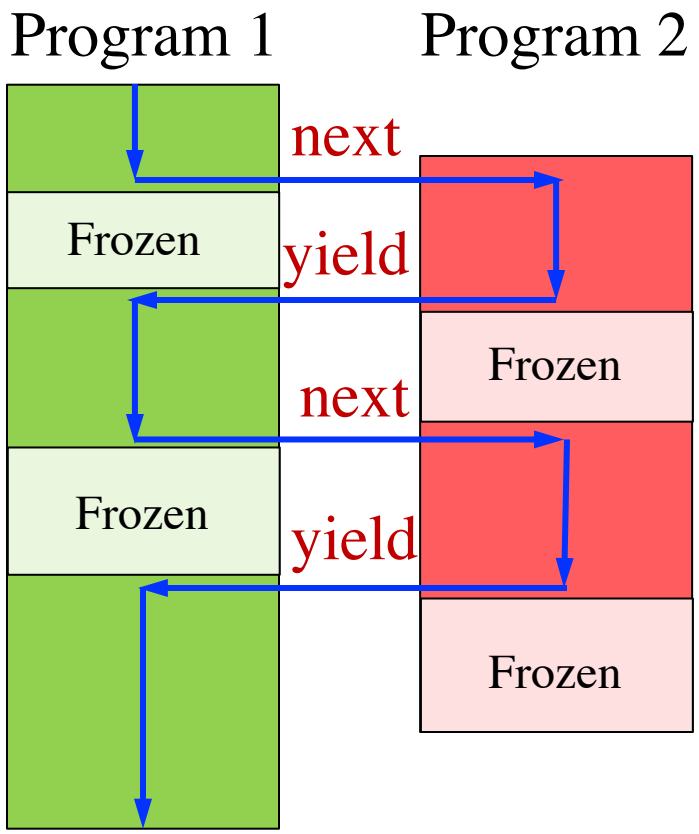


Subroutines vs Coroutines

Subroutine

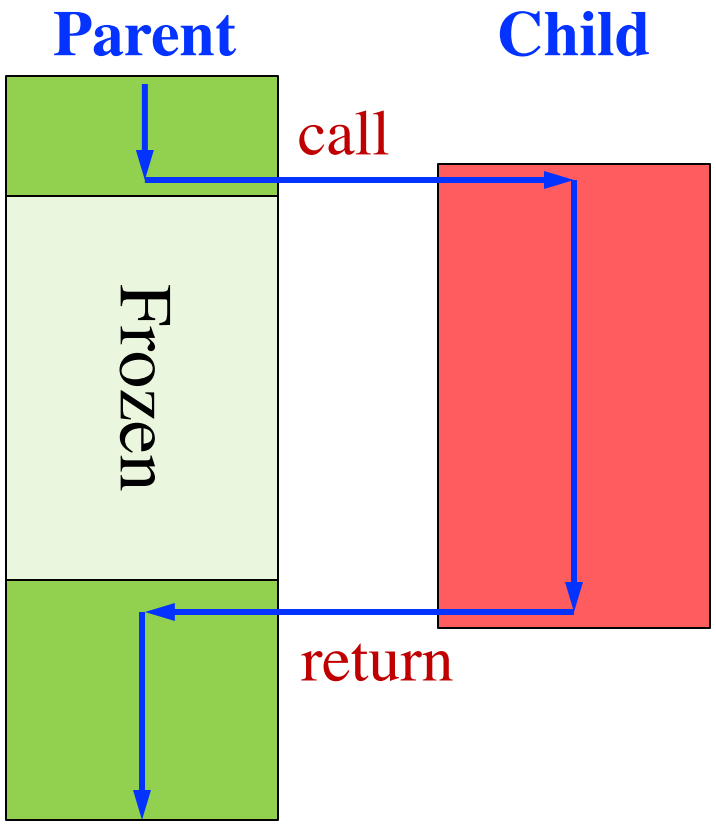


Coroutine

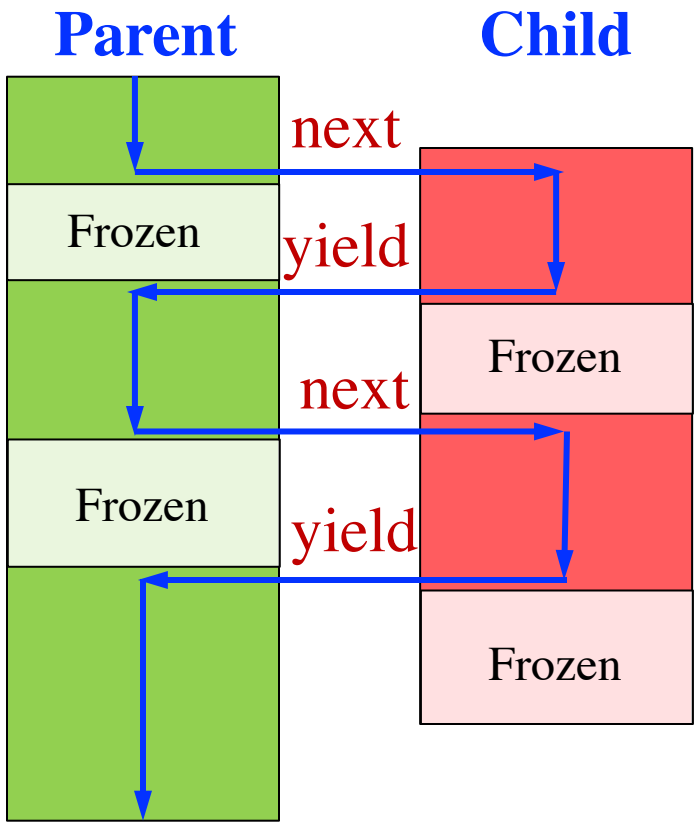


Subroutines vs Coroutines

Subroutine



Coroutine



Same Animation with Generator

```
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
```

Same Animation with Generator

```
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.i                f.image.angle+amount      # Update the angle
        if ab                 e-sangle) >= 90:
            se                nage.angle = fangle
            animating = False
        yield                                           # Pause to draw
```

Add this one line

Also Need to Drive 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:          # Something to animate
        try:
            next(self._animator)            # Move forward
        except StopIteration:
            self._animator = None           # Stop animating

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

Ignore input if still animating

Otherwise start animation for given input

Also Need to Drive The Animation

```
def update(self,dt):
    """Animates the image."""
    if not self._animator is None:        # Something to animate
        try:
            next(self._a                  nation forward
        except StopIte
            self._animato                 mating

    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
  - 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
  - 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
  - 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)

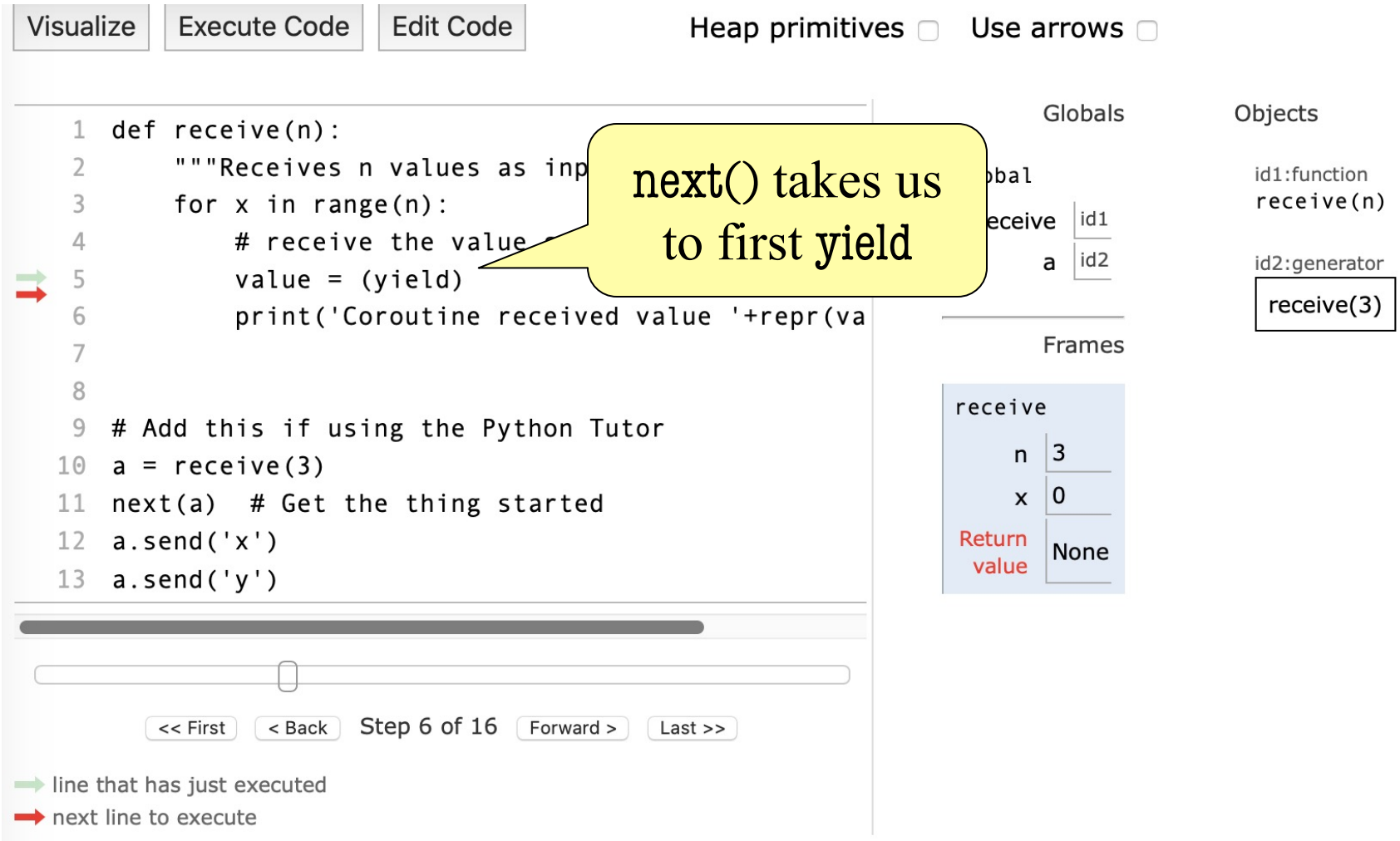
Generators Have a **send** Method

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

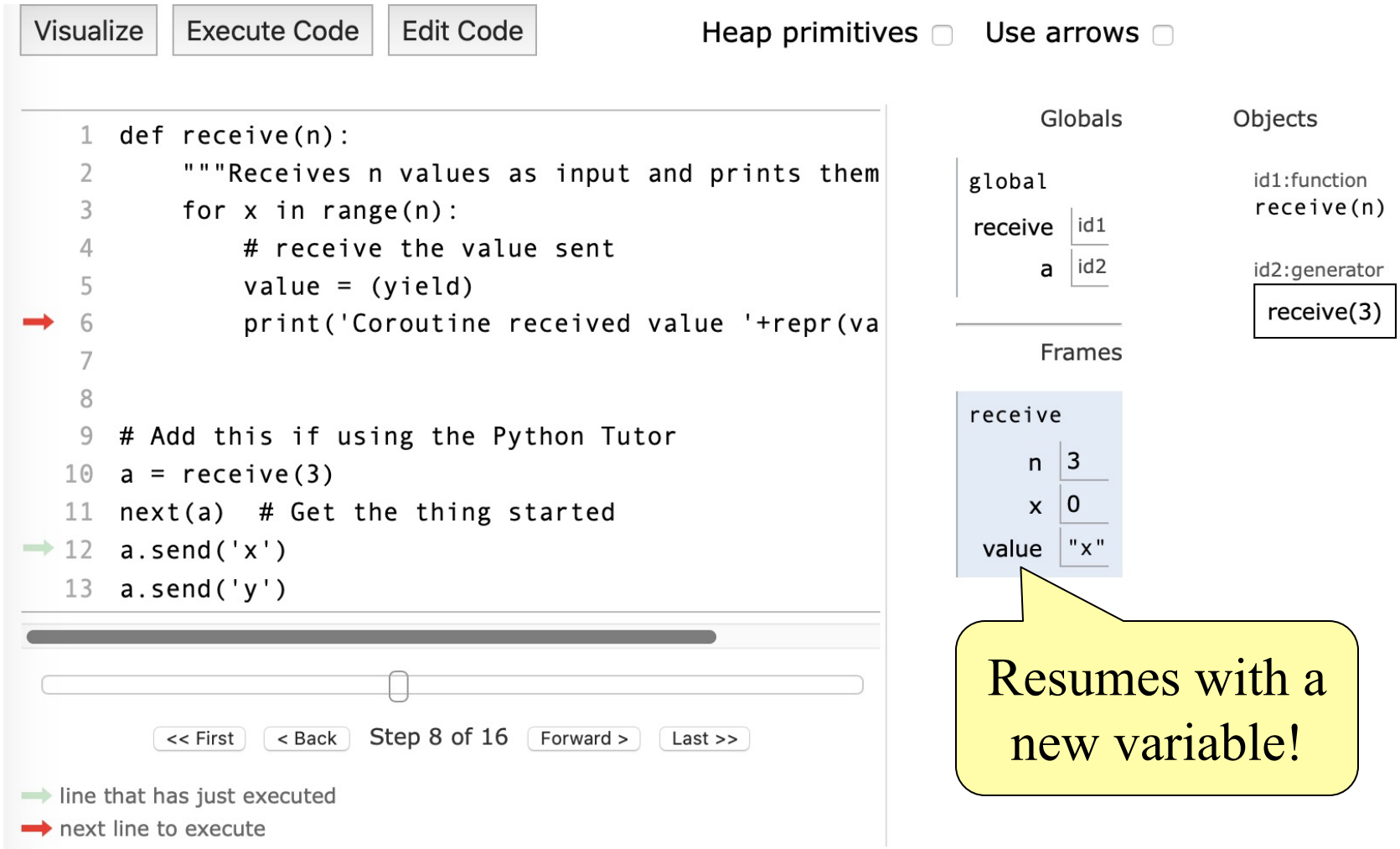
Must always start with next()

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

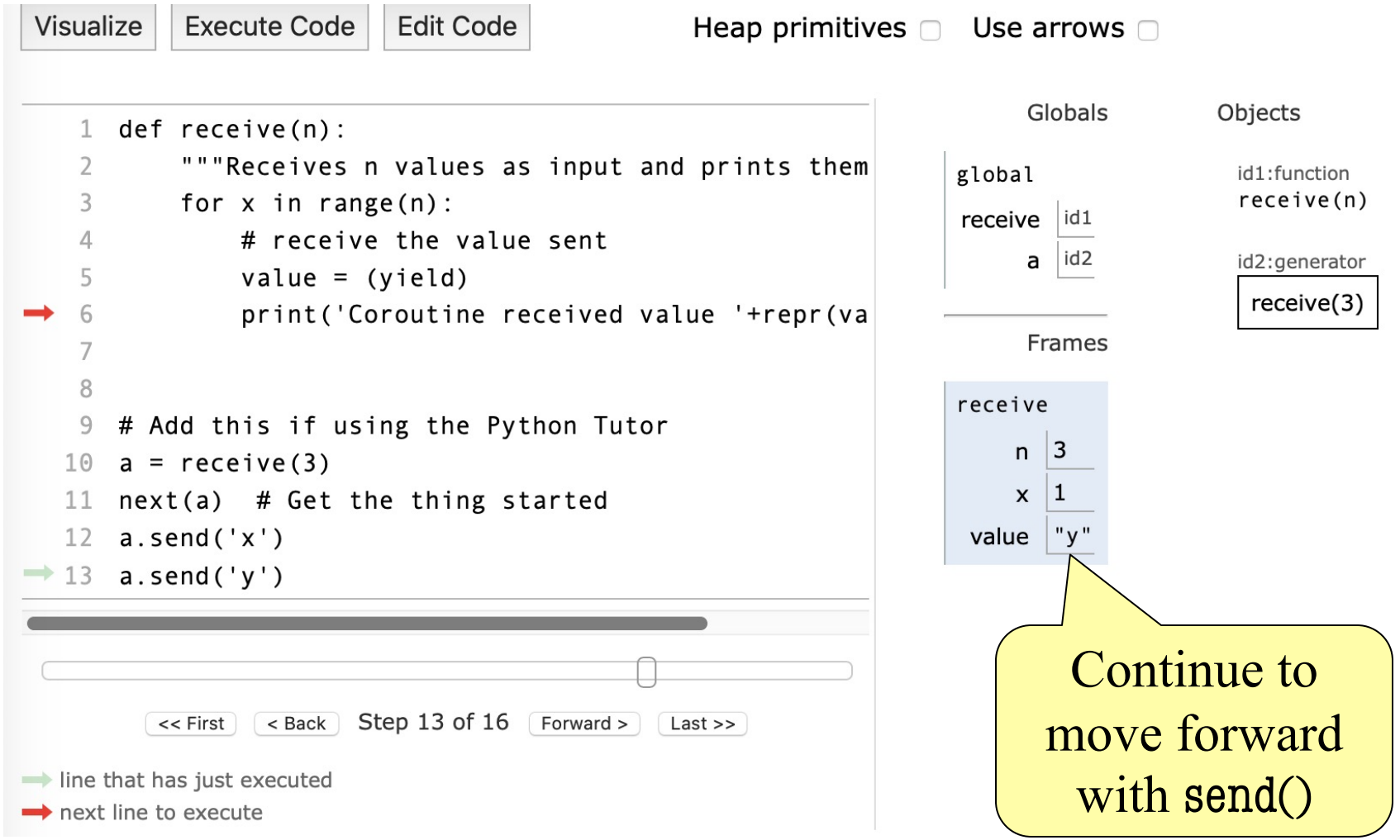
Visualizing in the Tutor



Visualizing in the Tutor



Visualizing in the Tutor

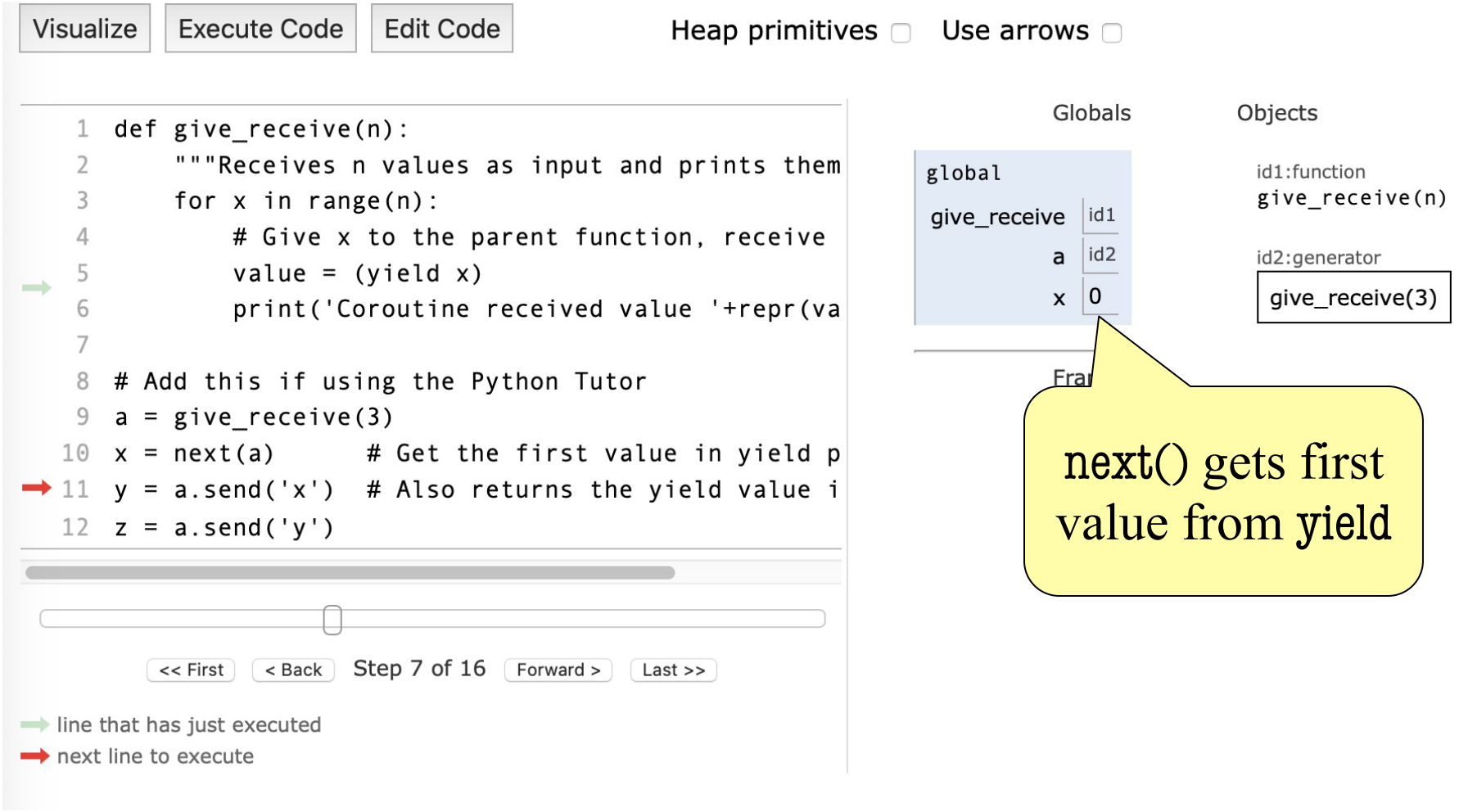


Can Do Both Ouput and Input

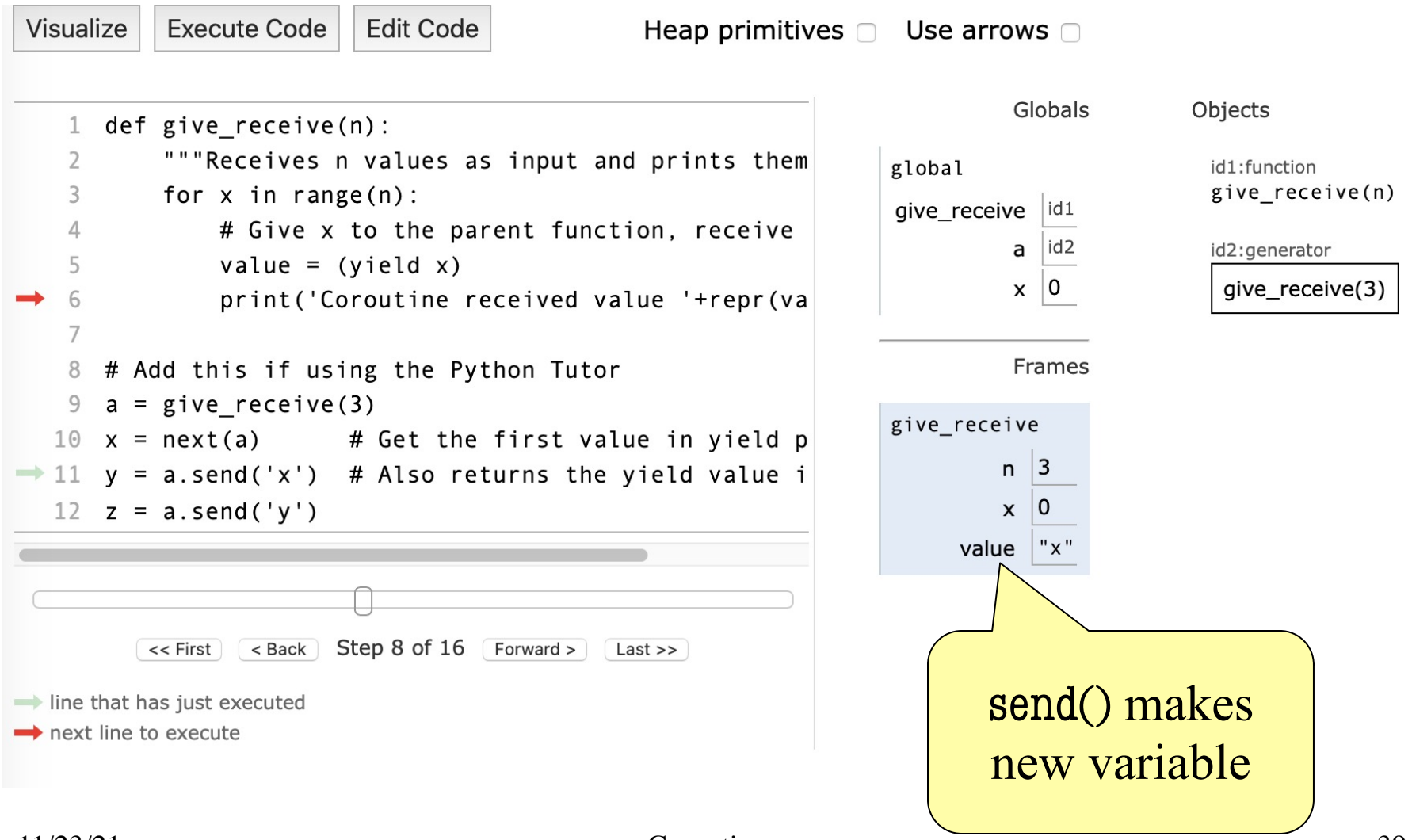
- **Format:** var = (yield expr)
  - Coroutine evaluates expr and outputs it
  - Coroutine stops and lets parent resume
  - When coroutine resumes, new value in var
- **Example:**

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

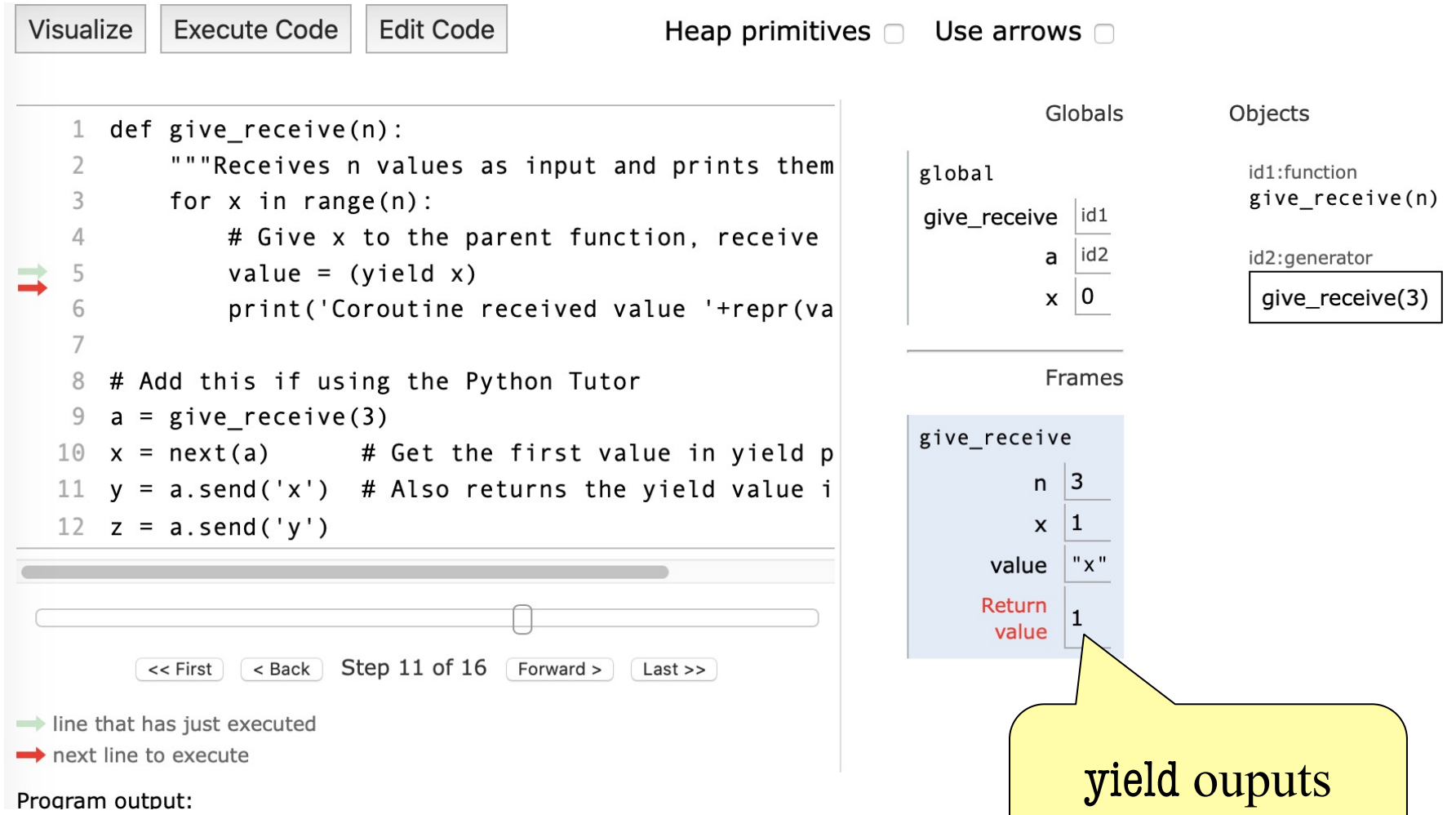
Visualizing Back-and-Forth



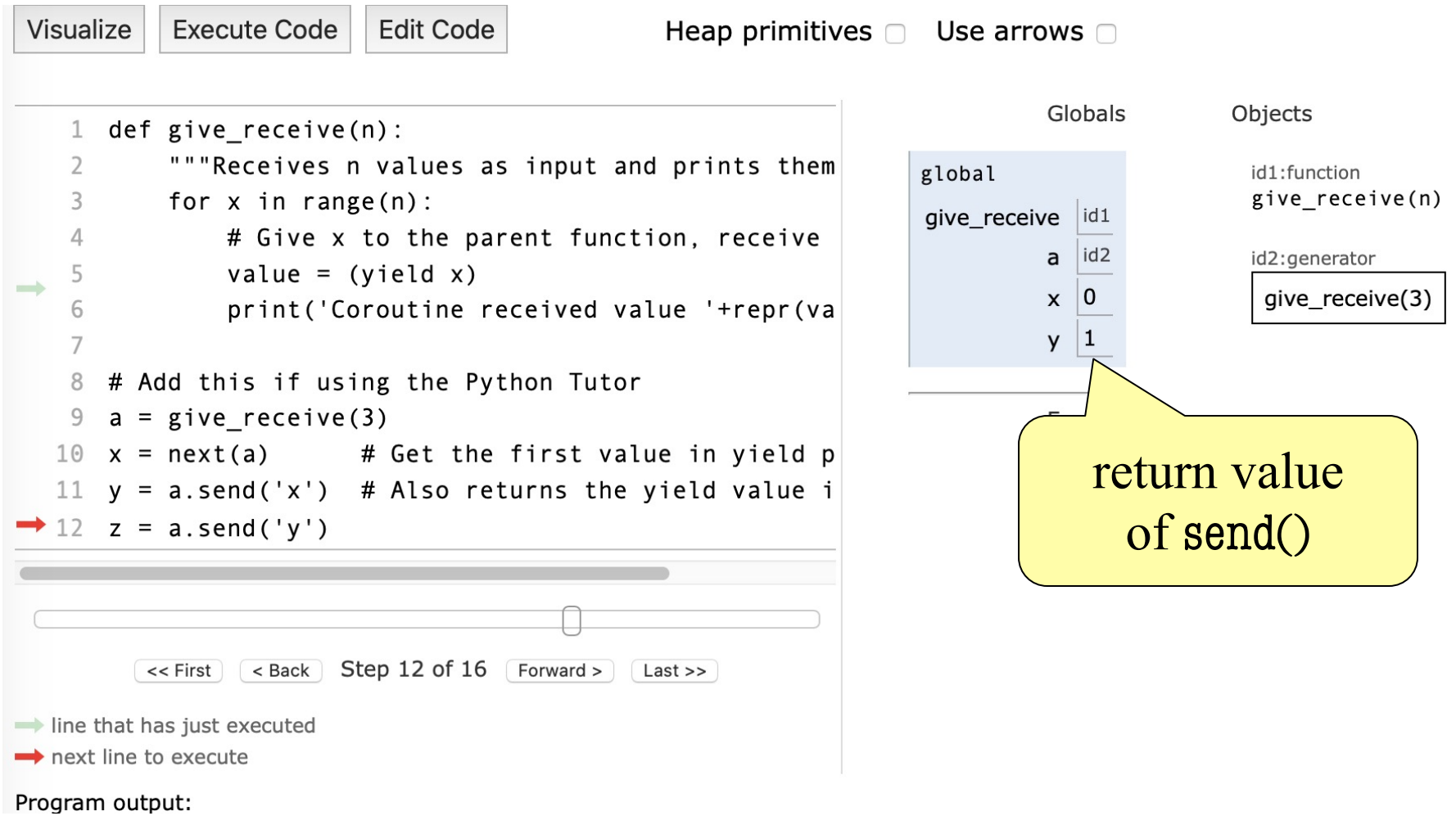
Visualizing Back-and-Forth



Visualizing Back-and-Forth



Visualizing Back-and-Forth



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`

Animation Smoothing 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
```

Animation Smoothing with Coroutines

```
def _animate_turn(self,direction):
    """Animates a rotation of the image over SPEED seconds"""
    sangle = self.image.angle
    fangle = sangle-                                    sangle-90
    steps = (fangle-                                    # Compute degrees per second
    animating = Tru
    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
```

Get the current dt as input each time

Parent Code Also Needs Tweaking

```
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
        ...
```

Parent Code Also Needs Tweaking

```
def update(self,dt):
    """Animates the image."""
    if not self._animator is None:        # Something to animate
        try:
            self._animator.send(dt)       # Send dt to the yield expression
        except:
            self._animator = None         # Stop animating
    elif self.input.is_key_down('left'):
        self._animator = self._animate_turn()
        next(self._animator)              # Start coroutine after creating it
        ...
```

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,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
```