Module 16

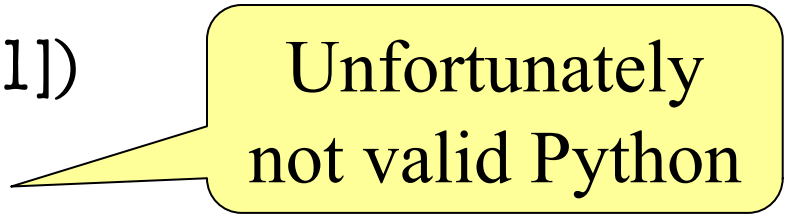
For-Loops

Motivating Example

```
def print_each(text):
    """Prints each character of text on a line by itself

    Example: print_each('abc') displays
        a
        b
        c

    Parameter text: The string to split up
    Precondition: text is a string"""
```

A First Attempt at the Function

```
def print_each(text):
    """Prints each character of text on a line by itself

    Precondition: text is a string """
    print(text[0])
    print(text[1])
    ...
    print(text[len(text)-1])
```



The Problem

- Strings are potentially **unbounded**
  - Number of characters inside them is not fixed
  - Functions must handle different lengths
  - **Example**: `print_each('a')` vs. `print_each('abcdfgh')`
- Cannot process with **fixed** number of lines
  - Each line of code can handle at most one element
  - What if # of elements > # of lines of code?
- We need a new **control structure**

The For-Loop

```
# Create local var x
x = text[0]
print(x)
x = text[1]
print(x)
...
x = text[len(text)-1]
print(x)
```

```
# Write as a for-loop
for x in text:
    print(x)
```

Key Concepts

- **iterable**: text
- **loop variable**: x
- **body**: print(x)

The For-Loop

```
# Create local var x
x = t
print
x = t
print(x)

...
x = text[len(text)-1]
print(x)
```

```
# Write as a for-loop
```

Iterable can be a string, tuple or list

Key Concepts

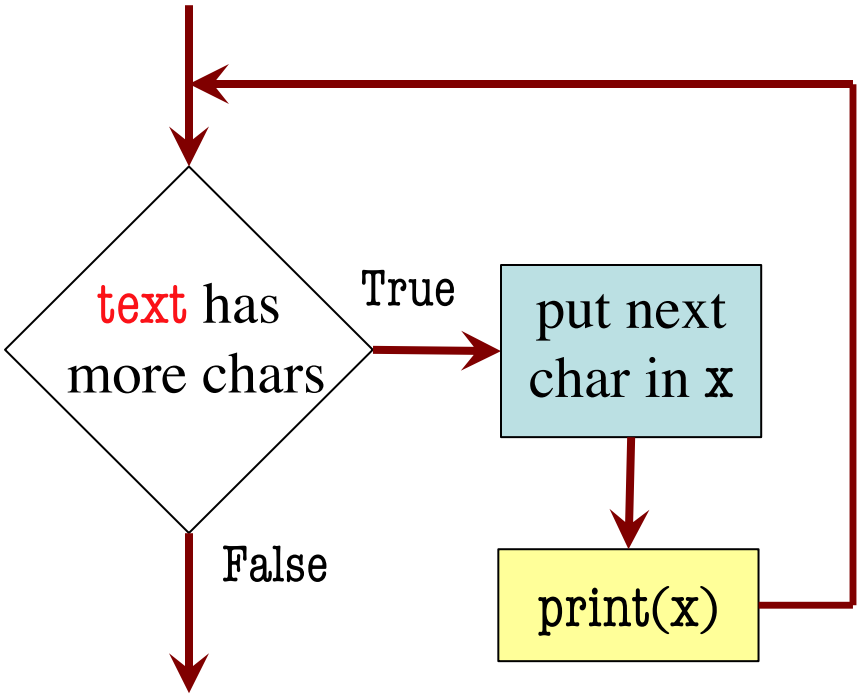
- **iterable**: text
- **loop variable**: x
- **body**: print(x)

Executing a For-Loop

The for-loop:

```
for x in text:
    print(x)
```

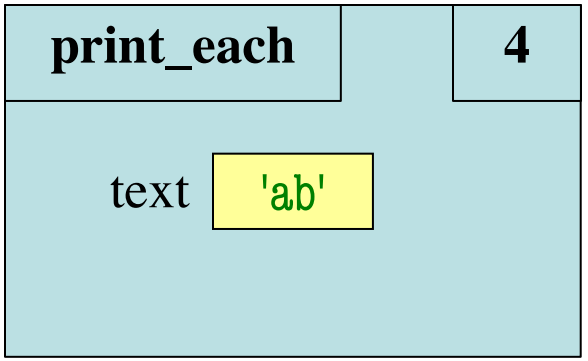
- **iterable:** text
- **loop variable**: x
- **body**: print(x)



For Loops and Call Frames

```
def print_each(text):
    """Prints each char of text
    Pre: text is a string"""
4   for x in thelist:
5       print(x)
```

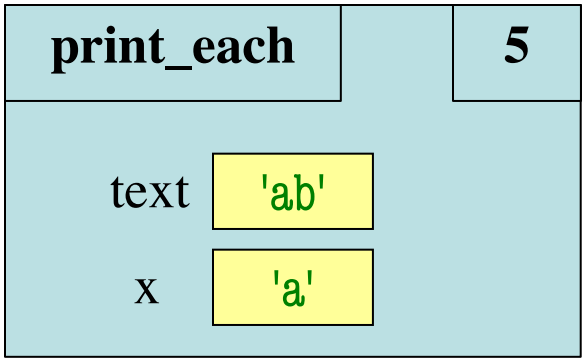
print_each(word):



For Loops and Call Frames

```
def print_each(text):
    """Prints each char of text
    Pre: text is a string"""
4   for x in thelist:
5       print(x)
```

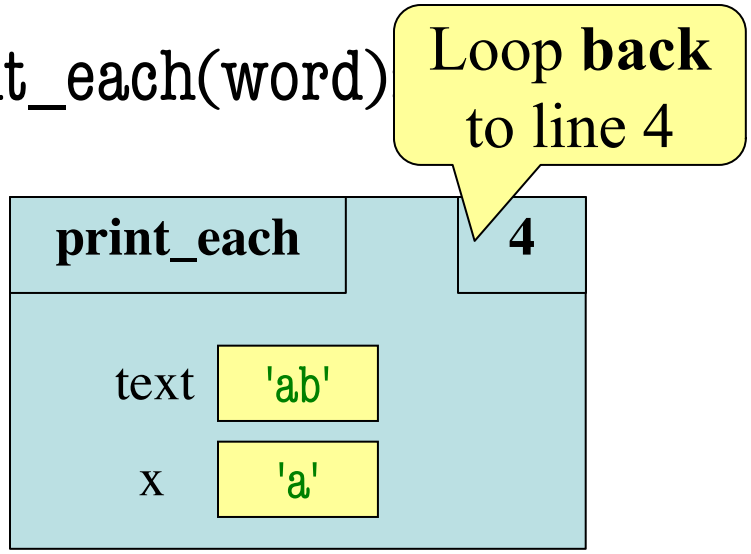
print_each(word):



For Loops and Call Frames

```
def print_each(text):
    """Prints each char of text
    Pre: text is a string"""
4   for x in thelist:
5       print(x)
```

print_each(word)

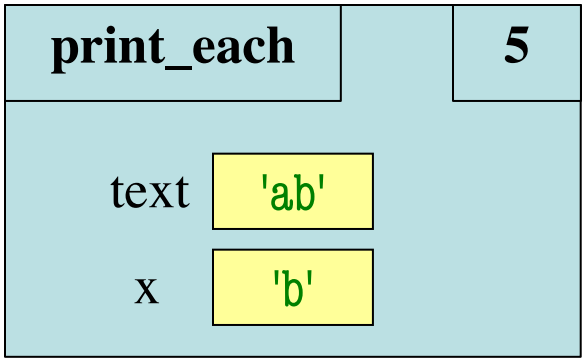


word 'ab'

For Loops and Call Frames

```
def print_each(text):
    """Prints each char of text
    Pre: text is a string"""
4   for x in thelist:
5       print(x)
```

print_each(word):



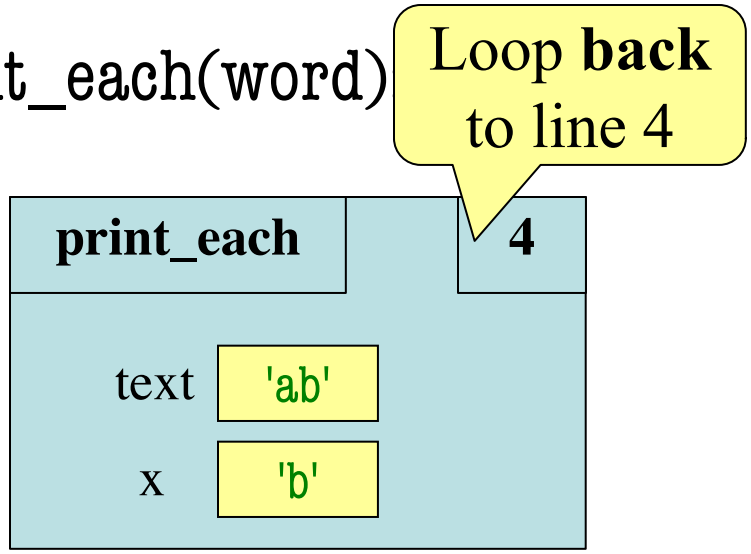
word 'ab'

Next element stored in x.
Previous value is lost.

For Loops and Call Frames

```
def print_each(text):
    """Prints each char of text
    Pre: text is a string"""
4   for x in thelist:
5       print(x)
```

print_each(word)

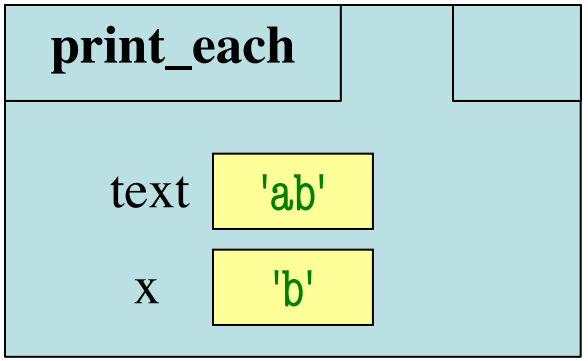


word 'ab'

For Loops and Call Frames

```
def print_each(text):
    """Prints each char of text
    Pre: text is a string"""
4   for x in thelist:
5       print(x)
```

print_each(word):



word 'ab'

Loop is **completed.**

Nothing new put in x.

For Loops and Call Frames

```
def print_each(text):
    """Prints each char of text
    Pre: text is a string"""
4   for x in thelist:
5       print(x)
```

print_each(word):

ERASE WHOLE FRAME

word | 'ab'

Example: Summing Elements of a Tuple

```
def sum(tups):
    """Returns: the sum of all elements in tups

    Precondition: tups is a tuple of all numbers
    (either floats or ints)"""
    pass # Stub to be implemented
```

Remember our approach:
Outline first; then implement

Example: Summing Elements of a Tuple

```
def sum(tups):
    """Returns: the sum of all elements in tups

    Precondition: tups is a tuple of all numbers
    (either floats or ints)"""
    # Create a variable to hold result (start at 0)
    # Add each tuple element to variable
    # Return the variable
```

Example: Summing Elements of a Tuple

```
def sum(tups):
    """Returns: the sum of all elements in tups

    Precondition: tups is a tuple of all numbers
    (either floats or ints)"""
    result = 0

    for x in tups:
        result = result + x

    return result
```

Accumulator

- **iterable:** tups
- **loop variable**: x
- **body**: result=result+x

For Loops and Conditionals

```
def num_ints(tups):
    """Returns: the number of ints in tups
    Precondition: tups is a tuple of any mix of types"""
    # Create a variable to hold result (start at 0)
    # for each element in the tuple...
        # check if it is an int
        # add 1 if it is
    # Return the variable
```

For Loops and Conditionals

```
def num_ints(tups):
    """Returns: the number of ints in tups
    Precondition: tups is a tuple of any mix of types"""
    result = 0
    for x in tups:
        if type(x) == int:
            result = result+1
    return result
```

Body

The Accumulator

- In a previous example saw the **accumulator**
  - Variable to hold a final (numeric) answer
  - For-loop added to variable at each step
- This is a common *design pattern*
  - Popular way to compute statistics
  - Counting, averaging, etc.
- It is not just limited to numbers
  - Works on every type that can be *added*
  - This means strings, lists and tuples!

Example: String-Based Accumulator

```
def despace(s):
    """Returns: s but with its spaces removed
    Precondition: s is a string"""
    # Create an empty string accumulator
    # For each character x of s
        # Check if x is a space
        # Add it to accumulator if not
```

Example: String-Based Accumulator

```
def despace(s):
    """Returns: s but with its spaces removed
    Precondition: s is a string"""
    result = ''
    for x in s:
        if x != ' ':
            result = result+x
    return result
```

Example: String-Based Accumulator

```
def reverse(s):
    """Returns: copy with s with characters reversed.
    Example: reverse('hello') returns 'olleh'
    Precondition: s is a (possibly empty string)"""
    # Create an empty tuple accumulator
    # For each character x of s
        # Add x to FRONT of accumulator
```

Example: String-Based Accumulator

```
def reverse(s):
    """Returns: copy with s with characters reversed.
    Example: reverse('hello') returns 'olleh'
    Precondition: s is a (possibly empty string)"""
    result = ''
    for x in s:
        result = x+result
    return result
```

Example: List-Based Accumulator

```
def copy_add_one(lst):
    """Returns: copy with 1 added to every element
    Precondition: lst is a list of all numbers
    (either floats or ints)"""
    # Create an empty tuple accumulator
    # For each element x of lst
        # Add 1 to value of x
        # Add x to the accumulator
```

Example: List-Based Accumulator

```
def copy_add_one(lst):
    """Returns: copy with 1 added to every element
    Precondition: lst is a list of all numbers
    (either floats or ints)"""
    copy = []  # accumulator
    for x in lst:
        x = x +1
        copy = copy + [x]
    return copy
```

Alternate Version

```
def copy_add_one(lst):
    """Returns: copy with 1 added to every element
    Precondition: lst is a list of all numbers
    (either floats or ints)"""
    copy = []  # accumulator
    for x in lst:
        x = x+1
        copy.append(x)      # add to end of copy
    return copy
```

Modifies accumulator

The Comparison

- They appear to be the same
- But first is less efficient (TURN ARROWS OFF)

- List accums are preferable for large data

Motivation: Repeat a Number of Times

```
def hello(n):
    """Prints 'Hello World' n times
    Precondition: n > 0 is an int."""
    pass # Stub to be implemented
```

Idea: Use a For-Loop

```
def hello(n):
    """Prints 'Hello World' n times
    Precondition: n > 0 is an int."""
    lst = [1, 2, ..., n]
    for x in lst:
        print('Hello World')
```

How do we do this step?

The Range Iterable

range(x)

- Creates an *iterable*
  - Can be used in a for-loop
  - Makes ints (0, 1, ... x-1)
- But it is not a tuple!
  - A **black-box** for numbers
  - Entirely used in for-loop
  - Contents of folder hidden

Example

```
>>> range(3)
range(0,3)
>>> for x in range(3)
...     print(x)
0
1
2
```

alt **id2**

Solving the Problem

```
def hello(n):
    """Prints 'Hello World' n times
    Precondition: n > 0 is an int."""
     for x in range(n):
        print('Hello World')
```

Uses of Range

- Can convert to list
  - Remember: iterable!

```
>>> list(range(4))
[0, 1, 2, 3]
```

- Best for handling ints
  - Statistical calculations
  - Computing n samples
- Or fixed repeats

```
def sum_squares(n):
    """
    Rets: sum of squares to n
    Prec: n is int > 0
    """
    total = 0

    for x in range(n):
        total = total + x*x
```

Accumulator

Two Main Variations

- `range(a,b)`
  - Generates (a,…,b-1)
  - Useful when do not want to start at 0
  - Requires that b > a
- `range(a,b,n)`
  - Generates (a,a+n,…,b-1)
  - "Counting by evens (or threes)"
  - n must be > 0

Motivation: Splitting by Position

```
def partition(s):
    """Returns: a list splitting s in two parts

    The 1st element of the tuple is chars in even
    positions (starting at 0), while the 2nd is odds.

    Examples:
        partition('abcde') is ['ace','bd']
        partition('aabb') is ['ab', 'ab']

    Precondition: s is a string."""
    pass # Stub to be implemented
```

PseudoCode

```
def partition(s):
    """Returns: a list splitting s in two parts
    Precondition: s is a string."""
    # Create accumulators for first & second parts
    # For each character in s
        # Determine if character is at odd or even pos
        # Add it to the correct accumulator
    # Return list with the two parts
```

Good Idea but Wrong

```
def partition(s):
    """Returns: a list splitting s in two parts
    Precondition: s is a string."""
    first = ''; second = ''
    for x in s:
        pos = s.find(x)
        if pos % 2 == 0:
            first = first + x
        else:
            second = second + x
    return [first,second]
```

What to do if x appears twice?

```
>>> partition('aabb')
['aabb','']
```

Getting Positions

- We want the positions!
  - So loop over the positions, not elements
  - If have position, can access with `s[pos]`
- Notice that range(n) starts at 0
  - This is first position of a string/list/tuple

```
lst = [5, 2, 7, 1]
pos = [0, 1, 2, 3]
```

- So use `range(len(lst))`

The Correct Approach

```
def partition(s):
    """Returns: a list splitting s in two parts
    Precondition: s is a string."""
    first = ''
    second = ''
    for pos in range(len(s)):
        if pos % 2 == 0:
            first = first + s[pos]
        else:
            second = second + s[pos]
    return [first,second]
```

Motivation: A Mutable Function

```
def add_one(lst):
    """(Procedure) Adds 1 to every element in the list
    Precondition: lst is a list of all numbers
    (either floats or ints)"""
```

- Accumulator pattern no longer relevant
  - Do not want to accumulate a **new** list
  - Want to modify the original list
- What is the right way to approach this?

A Motivating Function

```
def add_one(lst):
    """(Procedure) Adds 1 to every element in the list
    Precondition: lst is a list of all numbers
    (either floats or ints)"""
    for x in lst:
        x = x+1
    # procedure; no return
```

DOES NOT WORK!

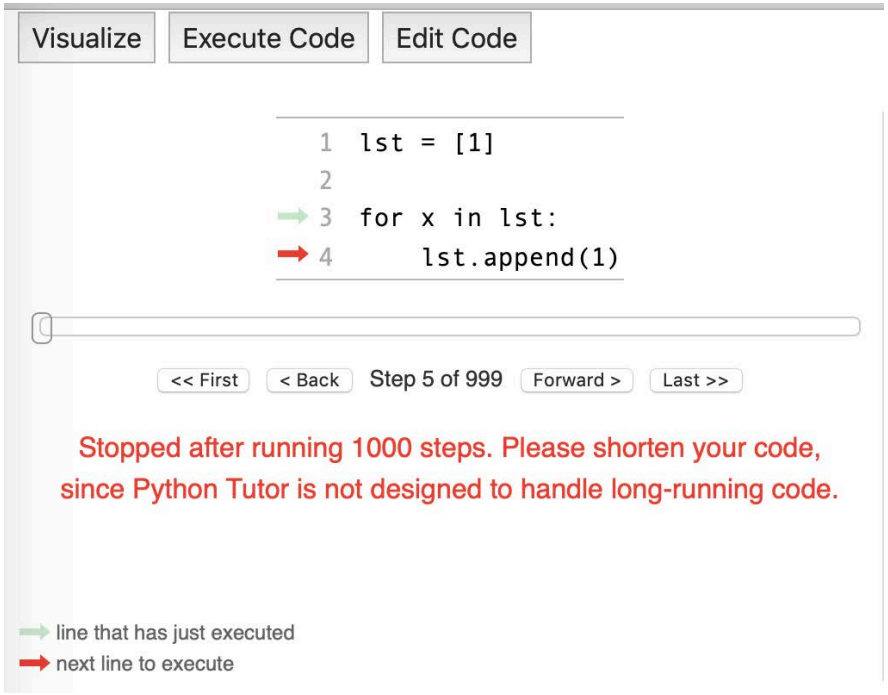
We need to put the answer into lst

Modifying a Loop Variable is Unsafe!

- This is an infinite loop:

```
for x in lst:
    lst.append(1)
```

- Best practices?
  - Never modify a loop var
  - Pick another iterable
  - Use that to modify first



Modifying the Contents of a List

```
def add_one(lst):
    """(Procedure) Adds 1 to every element in the list
    Precondition: lst is a list of all numbers
    (either floats or ints)"""
    size = len(lst)
    for k in range(size):
        lst[k] = lst[k]+1
    # procedure; no return
```

Iterator of list **positions** (safe)

WORKS!

Testing For-Loops

- Once again, we need code coverage
- But is automatic from **Rule of Numbers**
  - **Rule of 1:** Executes loop just once
  - **Rule of 2:** Executes loop many times
  - **Rule of 0:** Skips over loop entirely
- The hard part is what to do about **lists**
  - What if function is a mutable procedure?
  - What is the function is *accidentally* mutable?
- How do we have to adapt the test scripts?

Testing Immutable For-Loop

```
def copy_add_one(lst):
    """Returns: copy with 1 added to every element
    Precondition: lst is a list of all numbers
    (either floats or ints)"""
    ...
```

```
x = [1,2]
result = copy_add_one(x)
introcs.assert_equals([2,3],result)
introcs.assert_equals([1,2], x)
```

Verify the output (the **return value**)

Check that it is not accidentally **mutable**

Testing Mutable For-Loop

```
def add_one(lst):
    """(Procedure) Adds 1 to every element in the list
    Precondition: lst is a list of all numbers
    (either floats or ints)"""
    ...
```

```
x = [1,2]
result = add_one(x)
introcs.assert_equals([2,3],x)
introcs.assert_equals(None,result)
```

Verify the output (modified **argument**)

Check that it is not accidentally **fruitful**

Tuple Expansion

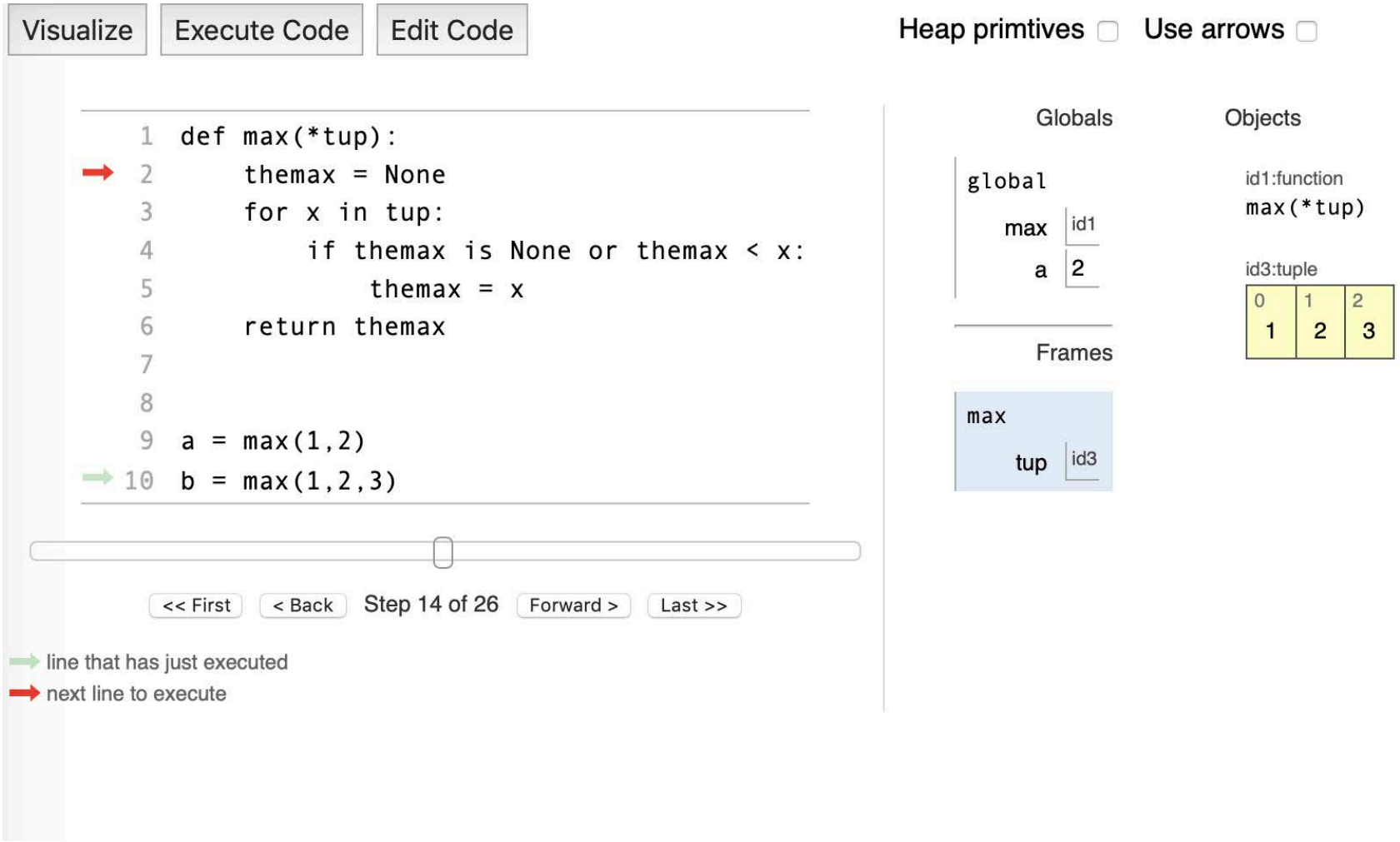
- Last use of lists/tuples is an advanced topic
  - But will see if read Python code online
  - Favored tool for data processing
- An Observation:
  - Function calls look like name + tuple
  - Why not pass a *single* argument: the tuple?
- Purpose of tuple expansion: *tuple
  - But only works in certain **contexts**

Tuple Expansion Example

```
>>> def add(x, y)
...     """Returns x+y """
...     return x+y
...
>>> a = (1,2)
>>> add(*a)          # Slots each element of a into params
3
>>> a = (1,2,3)      # Sizes much match up
>>> add(*a)
ERROR
```

Have to use in **function call**

Also Works in Function Definition



Also Works in Function Definition

```
def max(*tup):
    """Returns the maxi
    Param tup: The tuple of numbers
    Precond: Each element of tup is an int or float"""
    themax = None
    for x in tup:
        if themax == None or themax < x:
            themax = x
    return themax
```

Automatically converts all arguments to tuple