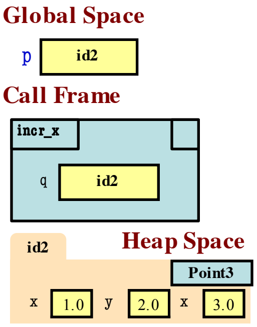
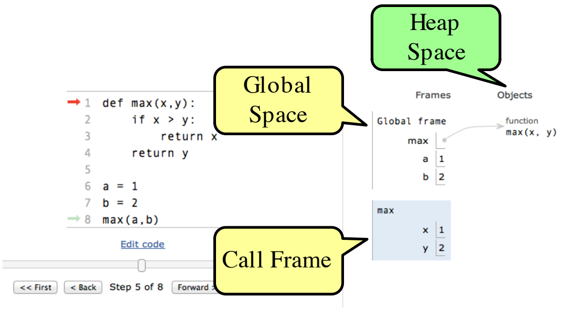
Modeling Storage in Python

- **Global Space**
  - What you "start with"
  - Stores global variables
  - Also **modules & functions!**
  - Lasts until you quit Python
- **Call Frame**
  - Variables in function call
  - Deleted when call done
- **Heap Space**
  - Where "folders" are stored
  - Have to access indirectly



Memory and the Python Tutor



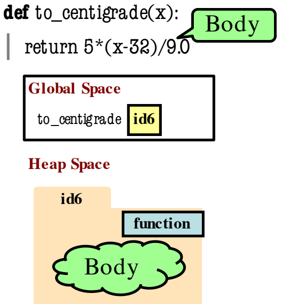
```
def max(x,y):
    if x > y:
        return x
    return y

a = 1
b = 2
max(a,b)
```

Functions and Global Space

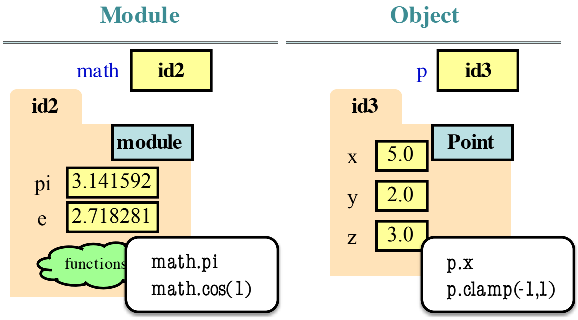
- A function definition…
  - Creates a global variable (same name as function)
  - Creates a **folder** for body
  - Puts folder id in variable
- Variable vs. Call

```
>>> to_centigrade
<fun to_centigrade at 0x100498de8>
>>> to_centigrade (32)
0.0
```

```
def to_centigrade(x):
    return 5*(x-32)/9.0
```



Modules vs Objects

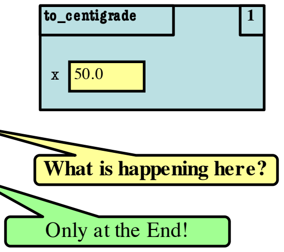


Recall: Call Frames

1. Draw a frame for the call
2. Assign the argument value to the parameter (in frame)
3. Execute the function body
   - Look for variables in the frame
   - If not there, look for global variables with that name
4. Erase the frame for the call

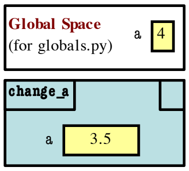
```
def to_centigrade(x):
    return 5*(x-32)/9.0
```

Call: to_centigrade(50.0)



Function Access to Global Space

- All function definitions are in some module
- Call can access global space for **that module**
  - math.cos: global for math
  - temperature.to_centigrade uses global for temperature
- But **cannot** change values
  - Assignment to a global makes a new local variable!
  - Why we limit to constants



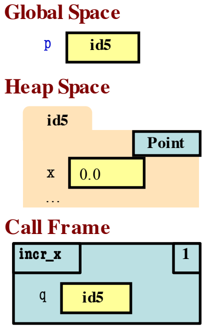
```
# globals.py
"""Show how globals work"""
a = 4 # global space

def change_a():
    a = 3.5 # local variable
```

Call Frames and Objects

- Mutable objects can be altered in a function call
  - Object vars hold names!
  - Folder accessed by both global var & parameter
- **Example**:

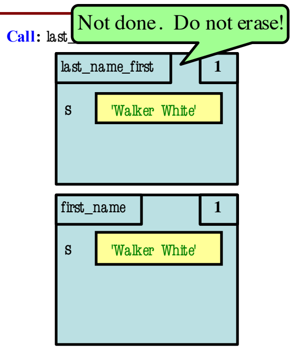
```
  def incr_x(q):
1 |   q.x = q.x + 1
```

```
>>> p = Point(0,0,0)
>>> incr_x(p)
```



Frames and Helper Functions

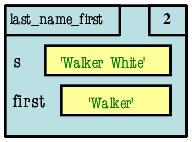
```
  def last_name_first(s):
    """Precondition: s in the form
    <first-name> <last-name>"""
1   first = first_name(s)
2   last = last_name(s)
3   return last + ',' + first

  def first_name(s):
    """Prec: see last_name_first"""
1   end = s.find(' ')
2   return s[0:end]
```



Frames and Helper Functions

```
  def last_name_first(s):
    """Precondition: s in the form
    <first-name> <last-name>"""
1   first = first_name(s)
2   last = last_name(s)
3   return last + ',' + first

  def first_name(s):
    """Prec: see last_name_first"""
1   end = s.find(' ')
2   return s[0:end]
```

Call: last_name_first('Walker White'):

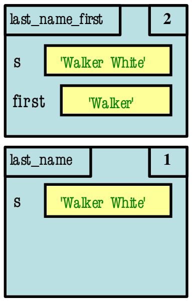


ERASE WHOLE FRAME

Frames and Helper Functions

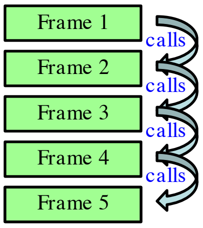
```
  def last_name_first(s):
    """Precondition: s in the form
    <first-name> <last-name>"""
1   first = first_name(s)
2   last = last_name(s)
3   return last + '.' + first

  def last_name(s):
    """Prec: see last_name_first"""
1   end = s.rfind(' ')
2   return s[end+1:]
```

Call: last_name_first('Walker White'):

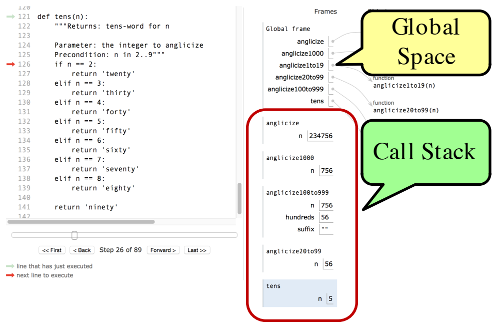


The Call Stack

- Functions are "stacked"
  - Cannot remove one above w/o removing one below
  - Sometimes draw bottom up (better fits the metaphor)
- Stack represents memory as a "high water mark"
  - Must have enough to keep the **entire stack** in memory
  - Error if cannot hold stack



Anglicize Example



```
def tens(n):
    """Returns: tens-word for n

    Parameter: the integer to anglicize
    Precondition: n in 2..9"""
    if n == 2:
        return 'twenty'
    elif n == 3:
        return 'thirty'
    elif n == 4:
        return 'forty'
    elif n == 5:
        return 'fifty'
    elif n == 6:
        return 'sixty'
    elif n == 7:
        return 'seventy'
    elif n == 8:
        return 'eighty'

    return 'ninety'
```

Frame	Variable	Value
anglicize	n	234756
anglicize1000	n	756
anglicize100to999	n	756
	hundreds	56
	suffix	""
anglicize20to99	n	56
tens	n	5