Lecture 9:
Memory in Python

CS 1110
Introduction to Computing Using Python

Text in fusia and extra slides were added after lecture for clarification. See slides 24 - 27.

[E. Andersen, A. Bracy, D. Fan, D. Gries, L. Lee, S. Marschner, C. Van Loan, W. White]
Announcements

• A1 revision process: A1 closed now on CMS for grading. Set your CMS notifications to “receive email when ...” When feedback is released, expected on Mar 13 afternoon, read resubmission instructions

• A2 to be released Thursday
Review: Nested Conditionals
Where is the robot?

• Angle of the robot relative to the sensor is $d$ degrees, where $d$ is non-negative

• Robot is in which quadrant?

• To avoid ambiguity, use this convention:
  - 1 if $0 \leq d < 90$
  - 2 if $90 \leq d < 180$
  - 3 if $180 \leq d < 270$
  - 4 if $270 \leq d < 360$

Can solve using if-elif-elif... Other options?
Nesting Conditionals

• Separate choices into 2 general categories
• Subdivide each category into subcategories
• Subdivide each subcategory further...

```python
if <above x-axis> :
    if <left of y-axis> :
        else:
    else:
        if <left of y-axis> :
            else:
See quadrants.py
```
Memory in Python
Global Space

- Global Space
  - What you “start with”
  - Stores global variables
  - Lasts until you quit Python

```
x = 4
```
Enter Heap Space

**Global Space**
- What you “start with”
- Stores global variables
- Lasts until you quit Python

**Heap Space**
- Where “folders” are stored
- Have to access indirectly

x = 4
p = shape.Point2(1,2)
q = shape.Point2(10,7)

p & q live in Global Space. Their folders live on the Heap.
Calling a Function Creates a Call Frame

What’s in a Call Frame?
• Boxes for parameters **at the start of the function**
• Boxes for variables local to the function **as they are created**

```python
def adjust_x_coord(pt, n):
    pt.x = pt.x + n

x = 4
p = shape.Point2(1,2)
adjust_x_coord(p, x)
```
Calling a Function Creates a Call Frame

What’s in a Call Frame?
• Boxes for parameters at the start of the function
• Boxes for variables local to the function as they are created

```python
def adjust_x_coord(pt, n):
    pt.x = pt.x + n

x = 4
p = shape.Point2(1,2)
adjust_x_coord(p, x)
```

```mermaid
graph LR
adjust_x_coord[adjust_x_coord]
pt[pt]
id1[id1]
n[4]
RETURN[None]
x[1]
y[2]
```

Global Space
- x: 4
- p: id1

Heap Space
- id1: Point2
- x: 1
- y: 2
Putting it all together

- **Global Space**
  - What you “start with”
  - Stores global variables
  - Lasts until you quit Python

- **Heap Space**
  - Where “folders” are stored
  - Have to access indirectly

- **Call Frames**
  - Parameters
  - Other variables local to function
  - Lasts until function returns
Two Points Make a Line

```
start = shape.Point2(0,0)
stop = shape.Point2(0,0)
print("Where does the line start?")
x = input("x: ")
start.x = int(x)
y = input("y: ")
start.y = int(y)
print("The line starts at ("+x+ ","+y+ ").")
print("Where does the line stop?")
x = input("x: ")
stop.x = int(x)
y = input("y: ")
stop.y = int(y)
print("The line stops at ("+x+ ","+y+ ").")
```

Where does the line start?
x: 1
y: 2
The line starts at (1,2).
Where does the line stop?
x: 4
y: 6
The line stops at (4,6).
Redundant Code is BAAAAAD!

```python
start = shape.Point2(0,0)
stop = shape.Point2(0,0)

print("Where does the line start?")
x = input("x: ")
start.x = int(x)
y = input("y: ")
start.y = int(y)
print("The line starts at ("+x+ ","+y+ ").")

print("Where does the line stop?")
x = input("x: ")
stop.x = int(x)
y = input("y: ")
stop.y = int(y)
print("The line stops at ("+x+ ","+y+ ").")
```
def configure(pt, role):
    print("Where does the line "+role+"?"")
    x = input("x:")
    pt.x = int(x)
    y = input("y:")
    pt.y = int(y)
    print("The line "+role+"s at ("+x+","+y+").")

start = shape.Point2(0,0)
stop = shape.Point2(0,0)
configure(start, "start")
configure(stop, "stop")
def configure(pt, role):
    print("Where does the line " + role + "?"")
    x = input("x: ")
    pt.x = int(x)
    y = input("y: ")
    pt.y = int(y)
    print("The line " +role+ "s at ("+x+ ","+y+ ")."")

start = shape.Point2(0,0)
stop = shape.Point2(0,0)
configure(start, "start")
configure(stop, "stop")
def get_coord(name):
    x = input(name+": ")
    return int(x)

def configure(pt, role):
    print("Where does the line " + role + "?"")
    pt.x = get_coord("x")
    pt.y = get_coord("y")
    print("The line " +role+ "s at ("+x+ ","+y+ ")." )

start = shape.Point2(0,0)
stop = shape.Point2(0,0)
configure(start, "start")
configure(stop, "stop")
Frames and Helper Functions

- Functions can call each other!
- Each call creates a *new call frame*
- Writing the same several lines of code in 2 places? Or code that accomplishes some conceptual sub-task? Or your function is getting too long? Write a **helper function**! Makes your code easier to
  - read
  - write
  - edit
  - debug
def get_coord(name):
    x = input(name +": ")
    return int(x)

def configure(pt, role):
    print("Where does the line " + role + "?"")
    pt.x = get_coord("x")
    pt.y = get_coord("y")
    print("The line " +role+ "s at ("+str(pt.x)+
      ","+str(pt.y)+ ")."")

start = shape.Point2(0,0)
configure(start, "start")
def get_coord(name):
    x = input(name + "": ")
    return int(x)

def configure(pt, role):
    print("Where does the line " + role + "?"")
    pt.x = get_coord("x")
    pt.y = get_coord("y")
    print("The line " +role+ "s at ","+str(pt.y)+ ").")
    start = shape.Point2(0,0)
    configure(start, "start")

A: Cross out the configure call frame.
B: Create a get_coord call frame.
C: Cross out the 4 in the call frame.
D: A & B
E: B & C
def get_coord(name):
    x = input(name + "": ")
    return int(x)

def configure(pt, role):
    print("Where does the line " + role + "?"")
    pt.x = get_coord("x")
    pt.y = get_coord("y")
    print("The line " +role+ "s at ("+str(pt.x)+ ","+str(pt.y)+ ")."")

start = shape.Point2(0,0)
configure(start, "start")
def get_coord(name):
    x = input(name + "": ")
    return int(x)

def configure(pt, role):
    print("Where does the line " + role + "?"")
    pt.x = get_coord("x")
    pt.y = get_coord("y")
    print("The line " +role+ "s at ("+str(pt.x)+
            ","+str(pt.y)+ ").\"")

start = shape.Point2(0,0)
configure(start, "start")
def get_coord(name):
    x = input(name + "": ")
    return int(x)

def configure(pt, role):
    print("Where does the line " + role + "?"")
    pt.x = get_coord("x")
    pt.y = get_coord("y")
    print("The line " + role + "s at (" + str(pt.x) + "," + str(pt.y) + ").")

start = shape.Point2(0,0)
configure(start, "start")
def get_coord(name):
    x = input(name+": ")
    return int(x)

def configure(pt, role):
    print("Where does the line " + role + "?"")
    pt.x = get_coord("x")
    pt.y = get_coord("y")
    print("The line " +role+ "s at ("+str(pt.x)+
    ","+str(pt.y)+ ")." )

start = shape.Point2(0,0)
configure(start, "start")
The Call Stack

• The set of function frames drawn in call order

• Functions frames are “stacked”
  ▪ Cannot remove one above w/o removing one below

• Python must keep the entire stack in memory
  ▪ Error if it cannot hold stack (“stack overflow”)
Errors and the Call Stack

```python
def get_coord(name):
    x = input(name+": ")
    return int(x1)

def configure(pt, role):
    print("Where does the line " + role + "?")
    pt.x = get_coord("x")
    pt.y = get_coord("y")
    print("The line " + role + "s at ("+x+","+y+ ")." )

start = shape.Point2(0,0)
configure(start, "start")
```

Where does the line start?

```
x: 1
Traceback (most recent call last):
  File "v3.py", line 19, in <module>
    configure(start, "start")
  File "v3.py", line 14, in configure
    pt.x = get_coord("x")
  File "v3.py", line 10, in get_coord
    return str(x1)
NameError: name 'x1' is not defined```
Q2: what does the call stack look like at this point in the execution of the code?

Choose 1 stack; in that stack cross out any frames that should have ended.
Modules and Global Space

**Import**

- Creates a global variable (same name as module)
- Puts variables, functions of module in a **folder**
- Puts folder id in the global variable

```python
>>> import math
```

<table>
<thead>
<tr>
<th>Global Space</th>
<th>Heap Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>math</td>
<td>id5</td>
</tr>
<tr>
<td>id5</td>
<td>module</td>
</tr>
<tr>
<td>pi 3.141592</td>
<td>e 2.718281</td>
</tr>
<tr>
<td>functions</td>
<td></td>
</tr>
</tbody>
</table>
Modules vs Objects

>>> import math
>>> math.pi

>>> p = shapes.Point3(5,2,3)
>>> p.x

Global Space

math  id5

p    id3

Heap Space

id5  math module

pi  3.141592  e  2.718281

functions

id3  Point3

x  5

y  2  z  3

34
Functions and Global Space

A function definition

- Creates a global variable (same name as function)
- Creates a folder for body
- Puts folder id in the global variable

```
INCHES_PER_FT = 12

def get_feet(ht_in_inches):
    return ht_in_inches // INCHES_PER_FT
```

Global Space

- `INCHES_PER_FT`: 12
- `get_feet`: `id6`

Heap Space

- `id6`
- `function`

Body
Function Definition vs. Call Frame

---

Global Space

```
INCHES_PER FOOT = 12

def get_feet(ht_in_inches):
    feet = ht_in_inches / INCHES_PER FOOT
    return feet

f = get_feet(68)
print("You are at least "+str(f)+" feet tall!")
```

Objects

- @id1: function
  - get_feet(ht_in_inches)

Frames

- get_feet
  - ht_in_inches: 68
  - feet: 5
  - return value: 5

Heap Space

(call definition goes here)

Call Frame

(memory for function call)

*It's alive!*
Storage in Python

- **Global Space**
  - What you “start with”
  - Stores global variables, modules & functions
  - Lasts until you quit Python

- **Heap Space**
  - Where “folders” are stored
  - Have to access indirectly

- **Call Frame Stack**
  - Parameters
  - Other variables local to function
  - Lasts until function returns
Don’t draw module folder, function folder

Folders that we do not draw:

- Module folder is created upon `import`, for example,
  ```python
  import math
  ```

- Function folder is created with `def` (the function header), for example,
  ```python
  def get_feet(height_in_inches):
  ```

Don’t draw those folders and the variables that store their ids; we only explained those folders to explain what you see in Python Tutor. Do not draw them.
Q3: what does the call stack look like at this point in the execution of the code?

def f3():
    print("f3")
def f2():
    print("f2")
f3()
f3()
f3()
def f1():
    print("f1")
f2()
f1()