Lecture 10

Memory in Python
Announcements For This Lecture

Assignment 1

• Work on your revisions
  ▪ Read feedback carefully
  ▪ Want done by tomorrow
  ▪ Partial credit after Wed.

• **Survey**: 676 responded
  ▪ Deadline is tomorrow
  ▪ **Avg Time**: 6.5 hours!
  ▪ **STD Dev**: 3.6 hours

More Assignments

• **Assignment 2 TONIGHT**
  ▪ Scan and submit online
  ▪ Upload before midnight
  ▪ **Late**: -10% per day
  ▪ No lates after THURS

• **Assignment 3** is posted
  ▪ Due week from Friday
  ▪ Before you go on Fall Break
  ▪ Graded when you get back
The Three “Areas” of Memory

Global Space

Call Stack

The Heap
Global Space

• This is the **area you “start with”**
  - First memory area you learned to visualize
  - A place to store “global variables”
  - Lasts until you quit Python

• What are **global variables**?
  - Any assignment not in a function definition
  - Also **modules & functions!**
  - Will see more on this in a bit
The Call Stack

- The area **where call frames live**
  - Call frames are created on a function call
  - May be several frames (functions call functions)
  - Each frame deleted as the call completes
- Area of volatile, temporary memory
  - Less permanent than global space
  - Think of as “scratch” space
- Primary focus of Assignment 2
Heap Space or “The Heap”

- Where the “folders” live
  - Stores *only* folders
- Can only access indirectly
  - Must have a variable with identifier
  - Can be in global space, call stack
- MUST have variable with id
  - If no variable has id, it is *forgotten*
  - Disappears in Tutor immediately
  - But not necessarily in practice
- Role of the *garbage collector*
Everything is an Object!

- Last time we saw that everything is an object
  - Must have a folder in the heap
  - Must have variable in global space, call stack
  - But ignore basic types (int, float, bool, str)
- Includes **modules** and **function definitions**!
  - Object is created by import
  - Object is created by def
  - Already seen this in Python Tutor
Modules and Global Space

- Importing a module:
  - Creates a global variable (same name as module)
  - Puts contents in a folder
    - Module variables
    - Module functions
  - Puts folder id in variable
- `from` keyword dumps contents to global space

```python
import math
```

---

**Global Space**

**Heap Space**

- `id5` variable
- `math` module
- `pi` constant: 3.141592
- `e` constant: 2.718281
- Functions
Modules vs Objects

Module

- math
- id2
  - pi: 3.141592
  - e: 2.718281
  - functions: math.pi, math.cos(1)

Object

- id3
  - p
    - Point3
      - x: 5.0
      - y: 2.0
      - z: 3.0
      - p.x
      - p.clamp(-1,1)
Modules vs Objects

Module

<table>
<thead>
<tr>
<th>id2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.14159</td>
</tr>
<tr>
<td>2.71828</td>
</tr>
<tr>
<td>math</td>
</tr>
<tr>
<td>functions</td>
</tr>
<tr>
<td>math.pi</td>
</tr>
<tr>
<td>math.cos(1)</td>
</tr>
</tbody>
</table>

Object

<table>
<thead>
<tr>
<th>id3</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
</tr>
<tr>
<td>point3</td>
</tr>
<tr>
<td>y 2.0</td>
</tr>
<tr>
<td>z 3.0</td>
</tr>
<tr>
<td>p.x</td>
</tr>
<tr>
<td>p.clamp(-1,1)</td>
</tr>
</tbody>
</table>

The period (.) means "go inside of the folder"
So Why Have Both?

- Question is a matter of program design
  - Some software will use modules like objects
- Classes can have many instances
  - Infinitely many objects for the Point3 class
  - Reason we need a constructor function
- Each module is a unique instance
  - Only one possibility for pi, cosine
  - That is why we import them
  - Sometimes refer to as singleton objects
So Why Have Both?

• Question is a matter of program design
  ▪ Some software will use modules like objects

• Classes can have **many instances**
  ▪ Infinitely many
  ▪ Rarely practical

• Each module is a unique instance
  ▪ Only one possibility for \( \pi \), \( \cos \)
  ▪ That is why we import them
  ▪ Sometimes refer to as **singleton** objects

Choice is an advanced topic beyond scope of this course
How About import *?

```python
1 from math import *
2 x = cos(1)
```

Ouch!
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

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

  ```
>>> to_centigrade
<fun to_centigrade at 0x100498de8>
>>> to_centigrade (32)
0.0
  ```
Working with Function Variables

• So function definitions are **objects**
  ▪ Function names are just variables
  ▪ Variable refers to a folder storing the code
  ▪ If you reassign the variable, it is lost

• You can assign them to other **variables**
  ▪ Variable now refers to that function
  ▪ You can use that **NEW** variable to call it
  ▪ Just use variable in place of function name
Example: **add_one**

```python
1 def add_one(x):
2     """Returns x+1"""
3     return x+1
4
5     global add_one
6     z = y(2)
```

Frame remembers the original name
Example: `add_one`

```python
1 def add_one(x):
2     """Returns x+1""
3     return x+1
4
5 y = add_one
6 z = add_one
```

Usage is an advanced topic beyond scope of this course

Frame remembers the original name
Why Show All This?

- Many of these are **advanced topics**
  - Only advanced programmers need
  - Will never need in the context of 1110
- But you might use them by *accident*
- **Goal: Teach you to read error messages**
  - Need to understand what messages say
  - Only way to debug your own code
  - This means understanding the **call stack**
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

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

Call: to_centigrade(50.0)
Aside: What Happens Each Frame Step?

- The instruction counter always changes
- The contents only change if
  - You add a new variable
  - You change an existing variable
  - You delete a variable
- If a variable refers to a mutable object
  - The contents of the folder might change
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

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

Call: `to_centigrade(50.0)`

What is happening here?
Function Access to Global Space

- Consider code to right
  - Global variable `a`
  - Function definition `get_a`
- Consider the call `get_a()`
  - Call frame to the right
  - What happens?

A: It crashes
B: Returns None
C: Returns 4
D: I don’t know

```python
# globals.py
"""Show how globals work"""

a = 4  # global space

def get_a():
    return a
```
Function Access to Global Space

- Consider code to right
  - Global variable a
  - Function definition get_a
- Consider the call get_a()
  - Call frame to the right
  - What happens?

A: It crashes
B: Returns None
C: Returns 4  CORRECT
D: I don’t know

```python
# globals.py
"""Show how globals work"""
a = 4  # global space
def get_a():
    return a
```
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
  - Makes a *new local variable*!
  - Why we limit to constants

```python
# globals.py
"""Show how globals work"""

a = 4  # global space

def get_a():
    return a
```

Global Space
(for globals.py)

```
Global Space
(for globals.py)
a

global space

call can access global space

cannot change values

why we limit to constants
```

```
# globals.py
"""Show how globals work"""

a = 4  # global space

def get_a():
    return a
```

```
get_a 6
```

```
Global Space
(for globals.py)
a

global space

call can access global space

cannot change values

why we limit to constants
```

```
# globals.py
"""Show how globals work"""

a = 4  # global space

def get_a():
    return a
```

```
get_a 6
```

```
Global Space
(for globals.py)
a

global space

call can access global space

cannot change values

why we limit to constants
```

```
# globals.py
"""Show how globals work"""

a = 4  # global space

def get_a():
    return a
```

```
get_a 6
```

```
Global Space
(for globals.py)
a

global space

call can access global space

cannot change values

why we limit to constants
```

```
# globals.py
"""Show how globals work"""

a = 4  # global space

def get_a():
    return a
```

```
get_a 6
```
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
  - Makes a *new local variable*!
  - Why we limit to constants

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

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

Global Space
(for globals.py)

```
a 4

change_a

a 3.5
```

10/1/19 Memory in Python
Frames and Helper Functions

1. def last_name_first(s):

   """Precond: s in the form 'first-name last-name' """

2. first = first_name(s)

3. last = last_name(s)

4. return last + ',' + first

Call: last_name_first('Walker White'):

8. def first_name(s):

   """Precond: see above""

9. end = s.find(' ')  

10. return s[0:end]
1. def last_name_first(s):
   
   """Precond: s in the form 'first-name last-name' ""

2. first = first_name(s)

3. last = last_name(s)

4. return last + ',' + first

8. def first_name(s):
   
   """Precond: see above""

9. end = s.find(' ') 

10. return s[0:end]
Frames and Helper Functions

1. def last_name_first(s):
   
   """Precond: s in the form 'first-name last-name' """

2. first = first_name(s)

3. last = last_name(s)

4. return last + ',' + first

Call: last_name_first('Walker White'):

```
last_name_first  4
s   'Walker White'
```

8. def first_name(s):

9. """Precond: see above"""

10. end = s.find(' ') 

11. return s[0:end]

Call: first_name('Walker White'):

```
first_name  11
s   'Walker White'
end   6
```
Frames and Helper Functions

1. `def last_name_first(s):
2.     """Precond: s in the form 'first-name last-name' """
3.     first = first_name(s)
4.     last = last_name(s)
5.     return last + ',' + first

Call: last_name_first('Walker White'):

    last_name_first 4
    s                'Walker White'

8. `def first_name(s):
9.     """Precond: see above"""
10.    end = s.find(' ')
11.    return s[0:end]

    first_name
    s                'Walker White'
    end              6
    RETURN          'Walker'
Frames and Helper Functions

1. `def last_name_first(s):`

   """Precond: s in the form 'first-name last-name' """

   first = first_name(s)
   last = last_name(s)
   return last + ',' + first

2. `def first_name(s):`

   """Precond: see above"""

   end = s.find(' ') 
   return s[0:end]

Call: `last_name_first('Walker White'):

<table>
<thead>
<tr>
<th>last_name_first</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>'Walker White'</td>
</tr>
<tr>
<td>first</td>
<td>'Walker'</td>
</tr>
</tbody>
</table>
1. def last_name_first(s):
  2.     '''Precond: s in the form 'first-name last-name' '''
  3.     first = first_name(s)
  4.     last = last_name(s)
  5.     return last + ',' + first
     ...
13. def last_name(s):
  14.     '''Precond: see above'''
  15.     end = s.rfind(' ')
  16.     return s[end+1:]

Call: last_name_first('Walker White'):

<table>
<thead>
<tr>
<th>last_name_first</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>'Walker White'</td>
</tr>
<tr>
<td>first</td>
<td>'Walker'</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>last_name</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>'Walker White'</td>
</tr>
</tbody>
</table>
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
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
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
The Call Stack

- Functions are **stacked**
  - Cannot remove one above without 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

Book adds a special “frame” called module. This is **WRONG**! Module is global space.
Anglicize Example

```python
120 def tens(n):
121     """Returns: tens-word for n
122     Parameter: the integer to anglicize
123     Precondition: n in 2..9"
124     if n == 2:
125         return 'twenty'
126     elif n == 3:
127         return 'thirty'
128     elif n == 4:
129         return 'forty'
130     elif n == 5:
131         return 'fifty'
132     elif n == 6:
133         return 'sixty'
134     elif n == 7:
135         return 'seventy'
136     elif n == 8:
137         return 'eighty'
138     return 'ninety'
```

Frames & Objects
- Global frame
- anglicize
  - anglicize(n)
  - anglicize1000
  - anglicize1to19
  - anglicize20to99
  - anglicize100to999
  - tens
- anglicize
  - n 234756
- anglicize1000
  - n 756
- anglicize100to999
  - n 756
- hundreds 56
- suffix "
- anglicize20to99
  - n 56
- tens
  - n 5
```python
120 def tens(n):
121     """Returns: tens-word for n
122     Parameter: the integer to anglicize
123     Precondition: n in 2..9"
124     if n == 2:
125         return 'twenty'
126     elif n == 3:
127         return 'thirty'
128     elif n == 4:
129         return 'forty'
130     elif n == 5:
131         return 'fifty'
132     elif n == 6:
133         return 'sixty'
134     elif n == 7:
135         return 'seventy'
136     elif n == 8:
137         return 'eighty'
138     return 'ninety'
```

Global Space

Call Stack