CSI 34: Scope
Announcements & Logistics

- **Lab 04 Feedback is out! Can you interpret TestResults.txt?**
- **HW 5** will due tonight @ 10pm
- **Lab 4 Part 2** due Wednesday/Thursday 10pm
- **Midterm reminders:**
  - **Review:** **Monday 3/11** from 7-9pm
  - **Exam** **Thurs 3/14** from 6-7:30pm OR 8-9:30pm
  - Both exam and review are in Bronfman Auditorium

Do You Have Any Questions?
Last Time: Aliasing

- Attempts to change immutable objects (e.g., strings) produce clones
  - Changes to clones do not affect originals
  - No aliasing!
- We can create aliases of mutable objects
  - Aliases refer to the same object, so changes to that object through any alias affect value that other aliases observe
- For the list data type, += is sneakily replaced by .append()
  - This mutates the list!

Goal was to demystify surprising behavior: nothing in computer science is magic!
Today's Plan

• **Scope**: variables, functions, objects have limited accessibility/visibility.
  
  • Understanding how this works helps us make decisions about where to define variables/functions/objects

Goal is to again demystify surprising behavior: nothing in computer science is magic!
What gets printed to the screen?

```python
a = 3
b = 4
def square(x):
    return x * x
c = square(a) + square(b)
c = pow(c, 0.5)
print(c)
```

What gets printed to the screen?
What gets printed to the screen?

```
a = 3
b = 4
def square(x):
    return x * x

c = square(a) + square(b)
c = pow(c, 0.5)
print(c)
```

5.0
What gets printed to the screen?

What if we make this change?

```python
a = 3
b = 4
def square(x):
    return x * x
c = square(a) + square(b)
c = pow(c, 0.5)
print(c)
```

What gets printed to the screen?
What gets printed to the screen?

```
a = 3
b = 4
def square(x):
    return x * x
c = square(a) + square(b)
c = pow(c, 0.5)
print(c)
```

Same output!

5.0
What gets printed to the screen?

What if we make this change?

```python
a = 3
b = 4
def square(a):
    return a * a
c = square(a) + square(b)
c = pow(c, 0.5)
print(c)
```

What gets printed to the screen?
What gets printed to the screen?

Not the same output

```
a = 3
b = 4
def square(x):
    return x * x
c = square(a) + square(b)
c = pow(c, 0.5)
print(c)
```

But also not an error!

Big Question: When we reuse variable names, how does Python know what a variable refers to?
Scope Diagram
• In Gladden & Carter "Mark Hopkins" refers to Mark Hopkins '1824, President of Williams College 1836-1872.

• In TCL, "Mark Hopkins" refers to Professor Mark Hopkins, who started working at Williams in 2022.
Let's see it in python!

```
mar_hop = 111119  # Mark Hopkins '1824 student ID number

def gladden():
    al = 223456  # Al’s student ID number
    ann = 287654  # Ann’s student ID number
    print(al, ann, mar_hop)

def carter():
    brady = 277777  # Brady’s student ID number
    blake = 288888  # Blake’s student ID number
    print(brady, blake, mar_hop)

def tcl():
    mar_hop = 998877  # Mark Hopkins '2022 student ID number
    casey = 212233  # Casey’s student ID number
    cleo = 233444  # Cleo’s student ID number
    print(casey, cleo, mar_hop)

if __name__ == '__main__':
    gladden()  # prints?
    carter()  # prints?
    tcl()  # prints?
```
Let's see it in python!

```python
scope.py

mar_hop = 111119 # Mark Hopkins '1824 student ID number

def gladden():
    al = 223456 # Al’s student ID number
    ann = 287654 # Ann’s student ID number
    print(al, ann, mar_hop)

def carter():
    brady = 277777 # Brady’s student ID number
    blake = 288888 # Blake’s student ID number
    print(brady, blake, mar_hop)

def tcl():
    mar_hop = 998877 # Mark Hopkins ‘2022 student ID number
    casey = 212233 # Casey’s student ID number
    cleo = 233444 # Cleo’s student ID number
    print(casey, cleo, mar_hop)

if __name__ == '__main__':
    gladden() # 223456  287654  111119
    carter()  # 277777  288888  111119
    tcl()     # 212233  233444  998877
```
Let's see it in python!

```python
scope.py
mar_hop = 111119  # Mark Hopkins '1824 student ID number

def gladden():
al = 223456  # Al's student ID number
ann = 287654  # Ann's student ID number
print(al, ann, mar_hop)

def carter():
brady = 277777  # Brady's student ID number
blake = 288888  # Blake's student ID number
print(brady, blake, mar_hop)
print(ann)
def tcl():
mar_hop = 998877  # Mark Hopkins '2022 student ID number
casey = 212233  # Casey's student ID number
cleo = 233444  # Cleo's student ID number
print(casey, cleo, mar_hop)
print(ann)

if __name__ == '__main__':
gladden()
carter()  # NameError: name 'ann' is not defined
tcl()
```

What if we print(ann) in dorm_b() or dorm_cs()?
Local Before Global

When python encounters a new term, like a variable or function name, it first looks locally, before looking higher up.

If python can't ever find the value assigned to the term, you get a NameError.
triple(num)
A Small Example
What will each of these print?
What will each of these print?

A
```python
def triple(num):
    multiplier = 3
    return multiplier * num
answer = triple(5)
print(answer)
```

B
```python
multiplier = 3
def triple(num):
    return multiplier * num
answer = triple(5)
print(answer)
```

C
```python
def triple(num):
    return multiplier * num
multiplier = 3
answer = triple(5)
print(answer)
```

D
```python
def triple(num):
    return multiplier * num
answer = triple(5)
multiplier = 3
print(answer)
```

NameError: name 'multiplier' is not defined
Function Frame Model
Scope: Function Frame Model

• By default, Python reads code one line at a time, starting from line 0

```python
0  multiplier = 3
1  def triple(num):
   2      return multiplier * num
2  answer = triple(5)
3  print(answer)
```
Scope: Function Frame Model

- At first, when variables are assigned, their values are stored in the global frame.

```python
0 multiplier = 3
1 def triple(num):
    return multiplier * num
2 answer = triple(5)
3 print(answer)
```
Scope: Function Frame Model

- Function definitions are treated like a single line of code
- A `def` statement does **not** call the function, it just defines it

```
0  multiplier = 3
1  def triple(num):
2      return multiplier * num
3  answer = triple(5)
4  print(answer)

Global Frame

multiplier: 3
triple: multiplier * num
```
Scope: Function Frame Model

• Function definitions are treated like a single line of code

• A `def` statement does not call the function, it just defines it

• Effectively, it assigns the name of the function to a blueprint for computing the function

```python
multiplier = 3

def triple(num):
    return multiplier * num

answer = triple(5)
print(answer)
```

Global Frame

multiplier : 3
triple :
Scope: Function Frame Model

- To execute an assignment statement, python first computes the value of the expression on the **right-hand side**

- In this case, the **right-hand side** calls the **triple** function

```python
0 multiplier = 3
1 def triple(num):
   return multiplier * num
2 answer = triple(5)
3 print(answer)
```

**Global Frame**

```
multiplier : 3
triple :
```
Scope: Function Frame Model

- When a function is called, a new frame is created to record the variables used by that function.

```
0 multiplier = 3
1 def triple(num):
    return multiplier * num
2 answer = triple(5)
3 print(answer)
```
Scope: Function Frame Model

• First, the values of the argument variables are recorded in the call frame.
Scope: Function Frame Model

• Then, the lines of the function are executed in order
• To look up the value of a variable, first python looks in the call frame

```python
multiplier = 3
def triple(num):
    return multiplier * num

answer = triple(5)
print(answer)
```

Global Frame
multiplier : 3
triple :

Call Frame
num : 5

? ? ? ?
Scope: Function Frame Model

• If the variable isn't found in the call frame, then Python looks in the parent frame
  • (the frame we were in when the function was called)

```python
multiplier = 3

def triple(num):
    return multiplier * num

answer = triple(5)
print(answer)
```

0 multiplier = 3
1 def triple(num):
   return multiplier * num
2 answer = triple(5)
3 print(answer)
Scope: Function Frame Model

- Ultimately, a return value is computed for the function call

```python
def triple(num):
    return multiplier * num

answer = triple(5)
print(answer)
```

Global Frame

multiplier : 3
triple :

Call Frame

num : 5

return value : 15
Scope: Function Frame Model

- The call frame is *destroyed* when the function returns

```python
multiplier = 3
def triple(num):
    return multiplier * num
answer = triple(5)
print(answer)
```

Global Frame
- `multiplier : 3`
- `triple :
  multiplier : 3`

Call Frame
- `num : 5`
  - `return value : 15`
Scope: Function Frame Model

- ...and the return value of the function call is assigned to variable `answer` in the global frame.
...and the return value of the function call is assigned to variable 

**answer** in the global frame

```python
multiplier = 3

def triple(num):
    return multiplier * num

answer = triple(5)
print(answer)
```

Global Frame:
- `multiplier`: 3
- `triple`:
  - `answer`: 15

Call Frame:
- `num`: 5
  - `return value`: 15
Scope: Function Frame Model

• Finally, the value of \texttt{answer} is looked up in the global frame
• And printed to the screen

```
0  multiplier = 3
1  def triple(num):
    \hspace{1em} return multiplier * num
2  answer = triple(5)
3  print(answer)
```

```
Global Frame

\begin{itemize}
  \item \texttt{multiplier} : 3
  \item \texttt{triple} : \
    \begin{itemize}
      \item \texttt{answer} : 15
    \end{itemize}
\end{itemize}
```

15
Function Frame Model: Side-by-Side Example
Let's use these principles to trace the execution of these two programs Side-By-Side.
def triple(num):
    return multiplier * num

multiplier = 3
answer = triple(5)
print(answer)
def triple(num):
    return multiplier * num

multiplier = 3
answer = triple(5)
print(answer)
def triple(num):
    return multiplier * num

multiplier = 3
answer = triple(5)
print(answer)

def triple(num):
    return multiplier * num

answer = triple(5)
multiplier = 3
print(answer)
```python
def triple(num):
    return multiplier * num
multiplier = 3
answer = triple(5)
print(answer)
```

Global Frame

```
def triple(num):
    return multiplier * num
answer = triple(5)
multiplier = 3
print(answer)
```

Call Frame
def triple(num):
    return multiplier * num

multiplier = 3
answer = triple(5)
print(answer)
def triple(num):
    return multiplier * num

multiplier = 3
answer = triple(5)
print(answer)
```python
def triple(num):
    multiplier = 3
    answer = triple(5)
    print(answer)
```

Global Frame

```
triple :
    multiplier : 3
```

Call Frame

```
num : 5
return value : 15
```
def triple(num):
    return multiplier * num

multiplier = 3
answer = triple(5)
print(answer)
def triple(num):
    return multiplier * num
multiplier = 3
answer = triple(5)
print(answer)
More Examples
What gets printed to the screen?

multiplier = 3
def mystery(num):
    return multiplier * num
multiplier = 2
answer = mystery(5)
print(answer)
What gets printed to the screen?

```python
multiplier = 3
def mystery(num):
    return multiplier * num
multiplier = 2
answer = mystery(5)
print(answer)
```

- `multiplier` is recorded as 3 on the Global Frame.
- Then the `mystery()` blueprint is recorded on the Global Frame.
- Then `multiplier` is re-assigned the value 2 on the Global Frame.
- `mystery(5)` evaluates to 10, since `multiplier` is 2 in the global frame and `num` is 5 in the call frame.
What gets printed to the screen?

```python
list = [2, 4, 6, 8]
list_str = list("whodoweappreciate")
print(list, list_str)
```
What gets printed to the screen?

```python
list = 2468
list_str = list("whodoweappreciate")
print(list, list_str)
```

```
TypeError: 'list' object is not callable
```

- `list` is a python keyword, in the Global Frame
- `list = ...` reassigns the value of list in the Global Frame
  - It's no longer the keyword, it's now an integer object
- So you can't call `list( .. )` as the built-in list-casting function!
  - ...This is why we should never use python keywords as variable names!!!
Helpful External Tool for Learning How python Executes Code:

The end!