Announcements

• Starting next week, submitting labs & attending section will provide a midterm safety net

• Homework 1 is due next Wednesday 1/28
  ▪ All homework is graded on effort; you must make progress on each problem to earn 2/2
  ▪ Homework Party on Tuesday 1/27 5–6:30pm in 2050 VLSB

• Quiz 1 released next Wednesday 1/28 is due next Thursday 1/29 (graded on correctness)

• Ask questions about lab and homework assignments in office hours! (cs61a.org/weekly.html)
  ▪ 2 locations in Bechtel Engineering Center (Map: http://goo.gl/dAcHXf)
  ▪ 11–2 & 3–5 on Monday, 11–6 on Tuesday & Thursday, 11–2 & 3–4 on Wednesday, 11–1 on Friday

• You need to register a class account (Lab 0); that's how we track assignments
  ▪ Please register even if you're on the waitlist or applying for concurrent enrollment
Names, Assignment, and User-Defined Functions

(Demo)
Types of Expressions

**Primitive expressions:**
- 2
- `add`
- 'hello'
  - **Number or Numeral**
  - **Name**
  - **String**

**Call expressions:**
- `max ( 2 , 3 )`
  - **Operator**
  - **Operand**
  - **Operand**

An operand can also be a call expression: `max(\text{min}(\text{pow}(3, 5), -4), \text{min}(1, -2))`
Discussion Question 1

What is the value of the final expression in this sequence?

```python
>>> f = min
>>> f = max
>>> g, h = min, max
>>> max = g
>>> max(f(2, g(h(1, 5), 3)), 4)
???
```
Environment Diagrams
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

**Code (left):**
- Statements and expressions
- Arrows indicate evaluation order

**Frames (right):**
- Each name is bound to a value
- Within a frame, a name cannot be repeated

(Demo)
Assignment Statements

Execution rule for assignment statements:

1. Evaluate all expressions to the right of $=$ from left to right.
2. Bind all names to the left of $=$ to those resulting values in the current frame.
Discussion Question 1 Solution

1  f = min
2  f = max
3  g, h = min, max
4  max = g
5  max(f(2, g(h(1, 5), 3)), 4)

Interactive Diagram
Defining Functions
Defining Functions

Assignment is a simple means of abstraction: binds names to values

Function definition is a more powerful means of abstraction: binds names to expressions

Function *signature* indicates how many arguments a function takes

```python
>>> def <name>(<formal parameters>):
    return <return expression>
```

Function *body* defines the computation performed when the function is applied

**Execution procedure for def statements:**

1. Create a function with signature `<name>(<formal parameters>)`
2. Set the body of that function to be everything indented after the first line
3. Bind `<name>` to that function in the current frame
Calling User-Defined Functions

Procedure for calling/applying user-defined functions (version 1):

1. Add a local frame, forming a new environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment

```
from operator import mul
def square(x):
    return mul(x, x)
square(-2)
```
Calling User-Defined Functions

Procedure for calling/applying user-defined functions (version 1):

1. Add a local frame, forming a new environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment

```
1 from operator import mul
2 def square(x):
  3    return mul(x, x)
4    square(-2)
```

Interactive Diagram

A function’s signature has all the information needed to create a local frame
Looking Up Names In Environments

Every expression is evaluated in the context of an environment.

So far, the current environment is either:
• The global frame alone, or
• A local frame, followed by the global frame.

Most important two things I’ll say all day:

An environment is a sequence of frames.

A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

E.g., to look up some name in the body of the square function:
• Look for that name in the local frame.
• If not found, look for it in the global frame.
  (Built-in names like “max” are in the global frame too, but we don’t draw them in environment diagrams.)
  (Demo)
Print and None

(Demo)
None Indicates that Nothing is Returned

The special value `None` represents nothing in Python.

A function that does not explicitly return a value will return `None`.

*Careful: None is not displayed* by the interpreter as the value of an expression.

```python
>>> def does_not_square(x):
...     x * x
...     # No return

The name sixteen is now bound to the value None

>>> does_not_square(4)  # None value is not displayed

>>> sixteen = does_not_square(4)
>>> sixteen + 4

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```
Pure Functions & Non-Pure Functions

**Pure Functions**
*just return values*

-2 \(\rightarrow\) \text{abs} \(\rightarrow\) 2

<table>
<thead>
<tr>
<th>Argument</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>2</td>
</tr>
</tbody>
</table>

2, 100 \(\rightarrow\) \text{pow} \(\rightarrow\) 126765060022829401496703205376

<table>
<thead>
<tr>
<th>2 Arguments</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2, 100</td>
<td>126765060022829401496703205376</td>
</tr>
</tbody>
</table>

**Non-Pure Functions**
*have side effects*

-2 \(\rightarrow\) \text{print} \(\rightarrow\) None

- A side effect isn't a value; it's anything that happens as a consequence of calling a function

- Python displays the output “–2”
Nested Expressions with Print

None, None  ▶️  print(...):

Does not get displayed

display “None None”

>>> print(print(1), print(2))
1
2
None None

1 ▶️  print(...):

None

display “1”

2 ▶️  print(...):

None

display “2”