Announcements

- Sign up for a lab and discussion section here, they are happening this week
- We will be making up Monday’s lecture on Friday. Info will be on Ed ASAP
- Lab 0 released, due Thursday (this includes the submission and assessment, two separate Gradescope assignments)
- No Homework this week
- Lecture recordings will be up on bCourses first (before the website)
- Ask lecture (live and recorded) questions in the Ed thread!
Tips for Success in CS 61A

• According to the course guide, expect to spend about 21.5 hours a week on CS 61A
  • Sometimes more, sometimes less; YMMV

• Utilize all of your resources!
  • OH, Instructor OH, Lab, Discussion, tutoring, etc.

• Ask for help and collaborate (when allowed)

• Start early

• Practice Practice Practice (and did I mention, practice?)
  • Intuition based topics, not memorization
Types of expressions

An expression describes a computation and evaluates to a value

\[
\begin{align*}
18 + 69 & \quad \frac{6}{23} & \quad \sin \pi & \quad \log_2 1024 \\
2^{100} & \quad f(x) & \quad \sqrt{3493161} & \quad \lim_{x \to \infty} \frac{1}{x} \\
7 \mod 2 & \quad \sum_{i=1}^{100} i & \quad | -1869| & \quad (69) \binom{18}
\end{align*}
\]
Call Expressions in Python

All expressions can use function call notation

(Demo)
Anatomy of a Call Expression

Evaluation procedure for call expressions:

1. Evaluate the operator and then the operand subexpressions
2. Apply the function that is the value of the operator to the arguments that are the values of the operands

Operators and operands are also expressions

So they evaluate to values
Evaluating Nested Expressions

\[ \text{mul}(\text{add}(4, \text{mul}(4, 6)), \text{add}(3, 5)) \]

\[ \text{mul}(28, 8) \]

\[ \text{mul}(4, 6) \]

\[ \text{add}(4, 24) \]

\[ \text{add}(3, 5) \]

\[ \text{mul}(4, 6) \]

\[ \text{mul}(4, 6) \]

\[ \text{mul}(4, 6) \]
Evaluating Nested Expressions

Expression tree

Operand subexpression

Value of subexpression

1st argument to mul

Value of the whole expression
Names, Assignment, and User-Defined Functions
Types of Expressions

**Primitive expressions:**
- 2
- `add`
- 'hello'

Number or Numeral | Name | String

An operand can also be a call expression

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

An operand can also be a call expression

`max(min(pow(3, 5), -4), 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 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)
```

Global frame

```
func max(...)
f

func min(...)
g

func max(...)
h

max
```

3
Break
Defining Functions
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

A function is a sequence of code that perform a particular task and can be reused easily

Using functions lets us abstract away sequences of computation

They take in some input (known as their arguments) and transform it into an output (the return value)

We can create functions using def statements. Their input is given in a function call, and their output is given by a return statement.
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

>>> 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
Return Values

The `return` keyword returns a value to whoever calls the functions and then exits the function.

Because the function is exited at the `return`, any code that comes after is not run.

Forgetting a `return` is really easy to do, but forgetting it will lead to no value being returned by your function.
Common Function Mistakes

```python
def add(x , y):
    return sum
    sum = x + y
```

- Code after the return is not being run. The final line should go before the return.

```python
def add(num1 , num2):
    sum = x + y
```

- The function takes in num1, num2 but the function body uses x, y. Also, there is no return.

```python
def add():
    sum = x + y
    return sum
```

- The function is using variables that don’t seem to exist. They should probably be parameters for the function.
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

```python
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

```python
from operator import mul

def square(x):
    return mul(x, x)

square(-2)
```

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)
Summary

- Programs consist of statements, or instructions for the computer, containing expressions, which describe computation and evaluate to values.
- Values can be assigned to names to avoid repeating computations.
- An assignment statement assigns the value of an expression to a name in the current environment.
- Functions encapsulate a series of statements that maps arguments to a return value.
- A def statement creates a function object with certain parameters and a body and binds it to a name in the current environment.
- A call expression applies the value of its operator, a function, to the value(s) or its operand(s), some arguments.