Attendance: https://tinyurl.com/sp18neil01

# CS61A Discussion 01

January 26, 2018

TA: Neil Shah (neilshah430@berkeley.edu)
Dis 149
OH:  Tue 5-6p 109 Morgan Hall

# Announcements

Stay on track

- Lab 00/01 due tonight @ 11:59pm
- Hog Checkpoint 1 (individual) due Monday, January 29
- Hog due Thursday, February 1
- HW 02 due Thursday, February 1 (super short!)

# Agenda

Overview

1. Names Review
2. Functions
3. Control
4. Environments Intro
5. Problems

# Names and Assignment

## Binding to Values

- Values can bind to names so that we can use those values easily later on
- When we evaluate names, we get the values they hold
- To bind names to values, we use assignment statements

**Examples**

```
x = 3          pi = 3.14          doubled_x = x * 2
```

**Execution rule for assignment statements:**

1. Evaluate all expressions to right of `=` from left to right
2. Bind all names to left of `=` to resulting values

# Concept Check

## Your Turn

What would be the output of the following code snippet in the Python interpreter?

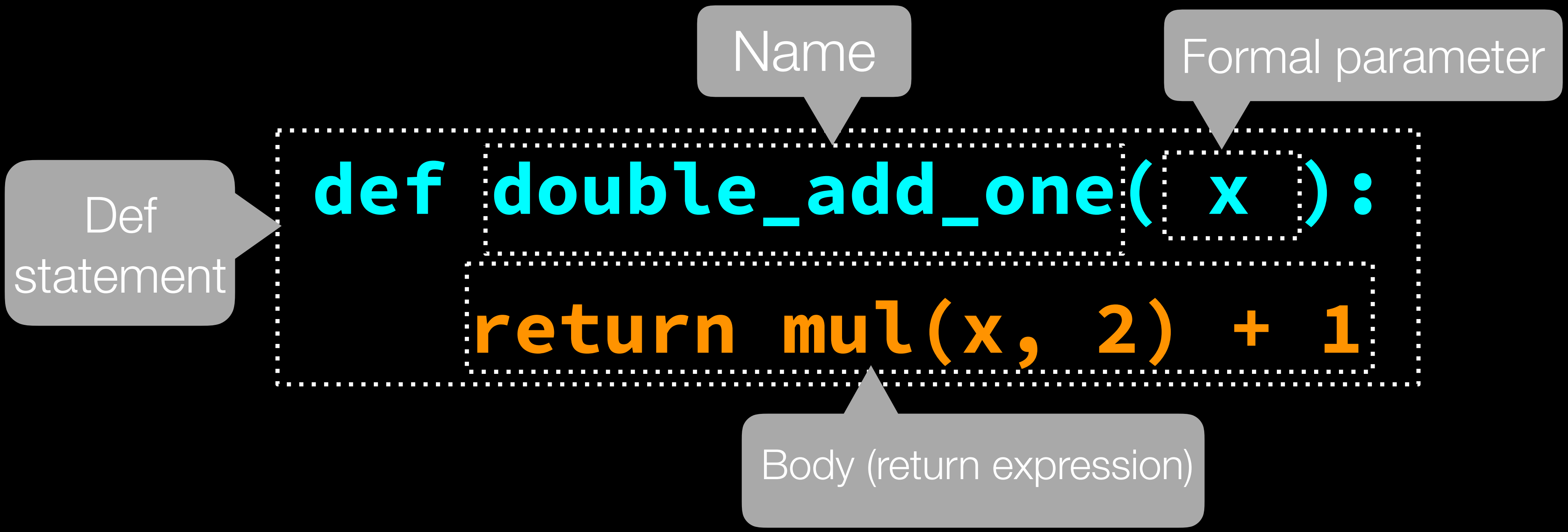
```
>>> b = 6
>>> double_b = b * 2
>>> b, double_b = double_b * 3, 5
>>> b
```

| Answer | Value |
|---|---|
| RED | 15 |
| ORANGE | 5 |
| GREEN | 12 |
| YELLOW | 36 |

# Functions

## Abstraction for Complexity

- Functions are abstractions for tasks, like squaring an input
- They give the user the ability to not have to worry about the underlying code, but rather focus on what the function does to an input to create an output
- Function name evaluates to a function, function call evaluates to return value of function



Name

Formal parameter

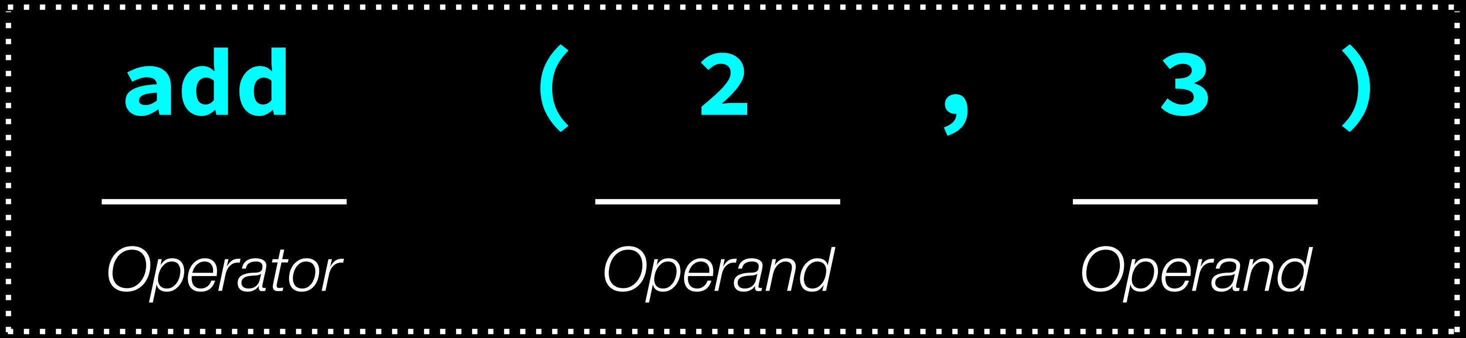
Def statement

```
def double_add_one( x ):
    return mul(x, 2) + 1
```

Body (return expression)

# Call Expressions

## Anatomy and Execution



```
add    ( 2 , 3 )
```

*Operator* *Operand* *Operand*

Operators, operands also expressions
and evaluate to values

**Procedure for call expressions:**

1. Evaluate operator, then operand subexpressions.

2. Apply function that is value of operator expressions
to arguments that are values of operand expressions

# Control Statements and Boolean Values

## Program Flow

- Statements are executed rather than evaluated and describe change to interpreter state
- Control statements control flow of program's execution based on logical comparisons
- Logical comparisons based on true/false values

**False-y Values**

False
None
0
[ ]
“”

**Truth-y Values**

everything else

# Conditional Statements

## Execution Order

- Consist of sequences of headers (usually the condition checks) and suites (code chunks executed if corresponding header has true value)
- Required **`if`** clause, optional (0 or more) **`elif`** clauses, and optional **`else`** clause

```
def absolute_value(x):
 “””Return the absolute value of x.”””
      if x < 0:
             return -x
      elif x === 0:
             return 0
      else:
             return x
```

Each clause is considered in order.

1. Evaluate header’s expression.
2. If true value, evaluate suite and skip remaining clauses.

# Iteration

## Repetition in a process

- What if we want to repeat execution of the same code segments over and over again? Do we just hardcode the chunk?
- Control statements let us express repetition in code
- **`while`** statement repeats code block as long as header condition remains true

```
def cube(x):
    return x*x*x

def mystery(x):
    total, count = 0, 0
    while count < x:
        total += x
        count += 1

    return total
```

If I execute **`cube(mystery(7))`** in the Python interpreter, what would output? How many **COMPLETE** iterations of the **`while`** loop?

| Answer | Value |
|---|---|
| RED | 42, 6 |
| ORANGE | 49, 7 |
| GREEN | 49, 8 |
| YELLOW | 42, 7 |

# Higher Order Functions

## Building Complexity with Function Arguments

Let’s say I want to have a function that sums up the first **n** natural numbers and another function that sums up the squares of the first **n** natural numbers.

```
def sum_naturals(n):
    total, k = 0, 1
    while k <= n:
      total, k = total + k, k + 1
    return total
```

```
def sum_squares(n):
    total, k = 0, 1
    while k <= n:
      total, k = total + k*k, k + 1
    return total
```

What’s the difference between the 2 functions?

# Higher Order Functions

## Building Complexity with Function Arguments

- Why not just use the same code for different, but similar processes?
- SOLUTION: Pass in a function as the argument!

```
def sum_generic(f, n):
    total, k = 0, 1
    while k <= n:
       total, k = total + f(k), k + 1
    return total
```

# Higher Order Functions

## Building Complexity with Functions Returned

- Returning functions from other functions is not meant to cause horror in your life. Berkeley already does enough of that!
- They are meant to simplify tasks for the user and allows the flexibility of knowing values of arguments later on

```
def compose(f, g):
    def work(x):
       return f(g(x))
    return work
```