## Lecture 2: Functions

Brian Hou
June 21, 2016

---

## Announcements

- Set up your computer and all accounts (Lab 0) by today
  - Piazza, Instructional (cs61a–??), OK
- Discussion sections begin today!
- Office hours begin today!
- Homework 0 is due tomorrow (Wednesday) at 11:59pm
- Quiz 1 will be on Thursday at the beginning of lecture

---

## Expressions

Primitive expressions, names, and environments

---

## Primitive expressions

- *Expressions* in programs evaluate to values
- *Primitive expressions* evaluate directly to values with minimal work needed
  - *Numbers* (e.g. 42, 3.14, 0)
  - *Names* (e.g. pi, add)
  - *Functions* (later today!)
- Some non-primitive expressions: 1 * 2, add(3, 4)

---

## Names                                         (demo)

- Giving names to values makes programming easier!
- An *assignment statement* is one way to bind a name to a value (e.g. x = 1)
- Each name can only be bound to one value
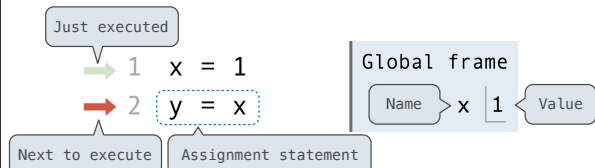  - *Environments* keep track of names and their values

**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 environment frame.

---

## Environment diagrams                          ([demo](demo))

- Environment diagrams visualize the interpreter's progress

Just executed

$\rightarrow$ 1  x = 1

$\rightarrow$ 2  y = x

Next to execute    Assignment statement

Global frame

Name  x  1  Value

| **Code (left)** | **Frames (right)** |
|---|---|
| Statements and expressions | Each name is bound to a value |
| | A name cannot be repeated in a frame |

## Functions

Call expressions, functions, and **def** statements

---

## Call expressions

$$add\ (\ 2\ ,\ 3\ )$$

operator ⎵ ⎵ operands

- *Call expressions* use functions to compute a value
- The operator and operands themselves are expressions
- To evaluate this call expression:
    1. *Evaluate* the operator to get a function value
    2. *Evaluate* the operands to get its values
    3. *Apply* the function to the values of the operands to get the final value

---

## Defining functions

- Functions have (inputs) and (outputs)

Function *signature* indicates name and number of arguments

**def** <name>(<parameters>):
    **return** <return expression>

Function *body* defines computation performed when function is applied

```
def square(x):
    return x * x
y = square(-2)
```
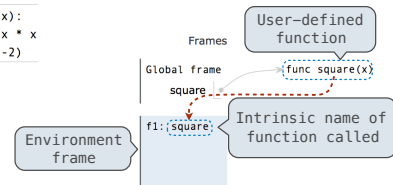
**Execution Rule for def Statements:**

1. Create a function with signature **<name>(<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

**Rules for calling user-defined functions (version 1):**

1. Create a new environment frame
2. Bind the function's parameters to its arguments in that frame
3. Execute the body of the function in the new environment

```
1 def square(x):
2     return x * x
3 y = square(-2)
```
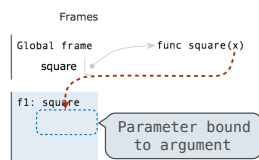
Frames

User-defined function

Global frame
square → func square(x)

f1: square

Intrinsic name of function called

Environment frame

---

## Calling user-defined functions

**Rules for calling user-defined functions (version 1):**

1. Create a new environment frame
2. Bind the function's parameters to its arguments in that frame
3. Execute the body of the function in the new environment

```
1 def square(x):
2     return x * x
3 y = square(-2)
```
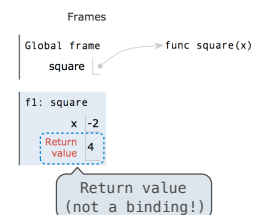
Frames

Global frame
square → func square(x)

f1: square

Parameter bound to argument

---

## Calling user-defined functions

**Rules for calling user-defined functions (version 1):**

1. Create a new environment frame
2. Bind the function's parameters to its arguments in that frame
3. Execute the body of the function in the new environment

```
1 def square(x):
2     return x * x
3 y = square(-2)
```

Frames

Global frame
square → func square(x)

f1: square
  x   -2
Return value   4

Return value (not a binding!)

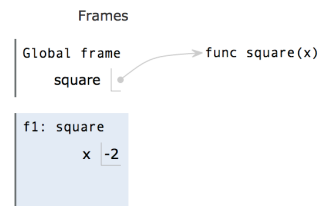## Break!

## Environments

## Looking up names in environments

- Every expression is evaluated in the context of an environment
- An environment is a sequence of frames
- So far, there have been two possible environments:
  - The global frame
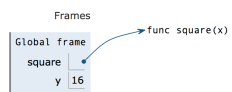  - A function's local frame, then the global frame

**Rules for looking up names in user-defined functions (version 1):**
1. Look it up in the local frame
2. If name isn't in local frame, look it up in the global frame
3. If name isn't in either frame, NameError

## Looking up names in environments



**Rules for looking up names in user-defined functions (version 1):**
1. Look it up in the local frame
2. If name isn't in local frame, look it up in the global frame
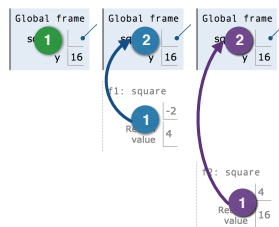3. If name isn't in either frame, NameError

## Multiple environments                    (demo)

```
>>> def square(x):
...     return x * x
>>> y = square(square(-2))
```



## None and Print

## None means that nothing is returned

- The special value None represents nothing in Python
- A function that does not explicitly return a value will return None
- *Note:* None is *not displayed* by the interpreter as the value of an expression

```
>>> def does_not_square(x):
...     x * x
>>> does_not_square(-2)
>>> not_four = does_not_square(-2)
>>> not_four + 4
TypeError: unsupported operand type(s) for +:
        'NoneType' and 'int'
```
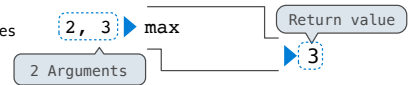
No **return**

The name **not_four** is now bound to the value **None**
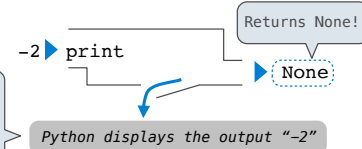
**None** value is not displayed

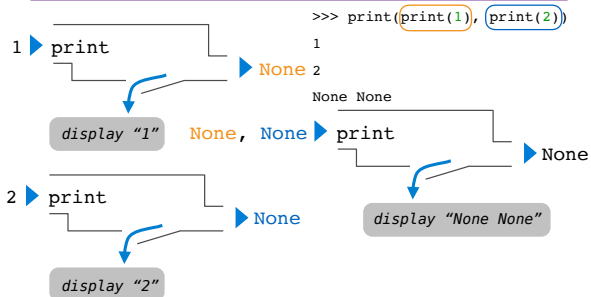## Pure and non-pure functions

*Pure functions* just return values

2, 3 ▶ max

2 Arguments

Return value

3

*Non-Pure functions* have side effects

-2 ▶ print

Returns None!

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

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

1 ▶ print

None

*display "1"*

None, None ▶ print

None

*display "None None"*

2 ▶ print

None

*display "2"*

## More Functions                                      (demo)

- The operands of a call expression can be any expression
- What about the expression **square**?

```
>>> four = describe(square, -2)
Calling function with argument -2
Result was 4
>>> four
4
>>> sixteen = describe(square, four)
Calling function with argument 4
Result was 16
>>> sixteen
16
```