

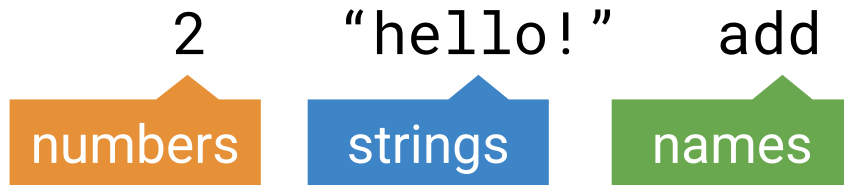
Lecture 2 - Names & Functions

2022 / 9 / 21

Program Structure

Review - Expressions

Primitive Expressions:



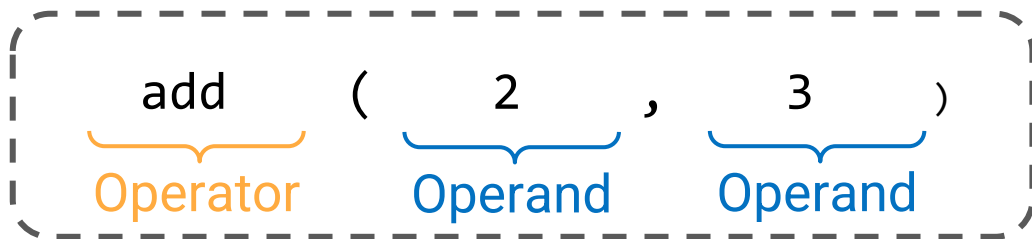
Arithmetic Expressions:

`1 + 2` `15 // 3`

Call Expressions:

`add(3, 4)`
`max(add(2, 3), 5 * min(-1, 4))`

Review - Evaluating Call Expressions



1. Evaluate

- a. Evaluate the operator subexpression
- b. Evaluate each operand subexpression

2. Apply

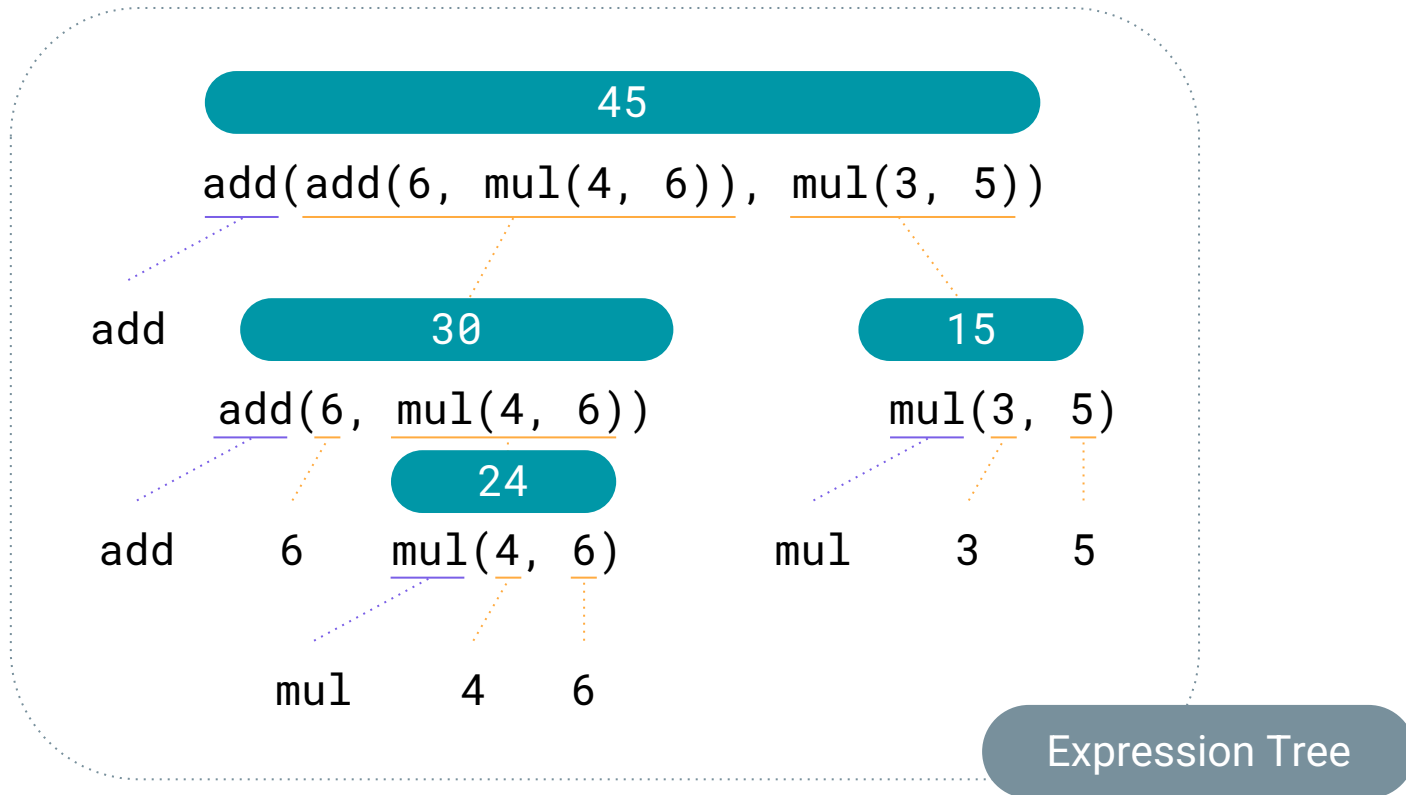
- a. Apply the value of the operator subexpression to the values of the operand subexpression

Nested Call Expression

1 Evaluate operator

2 Evaluate operands

3 Apply!



Values

Programs manipulate **values**

Values represent different **types** of data

Integers: 2 44 -3

Strings: "hello!" "cs61a"

Floats: 3.14 4.5 -2.0

Booleans: True False

Expressions & Values

Expressions **evaluate** to values in one or more steps

Expression:

Value:

`'hello!'`  `'hello!'`

`7 / 2`  `3.5`

`add(1, max(2, 3))`  `4`

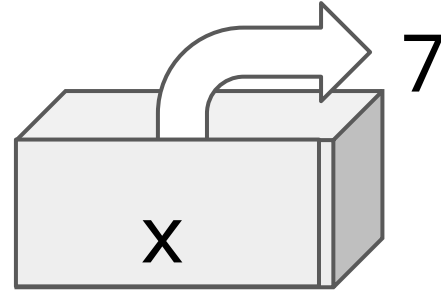
Names

Demo

Values can be assigned to **names** to make referring to them easier.

A name can only be bound to a single value.

One way to introduce a new name in a program is with an **assignment statement**.



x = 1 + 2 * 3 - 4 // 5

Name

Expression

Statements affect the program, but do not evaluate to values.

Check Your Understanding

```
>>> f = min
```

```
>>> f = max
```

```
>>> g, h = min, max
```

```
>>> max = g
```

```
>>> max(f(2, g(h(1, 5), 3)), 4)
```

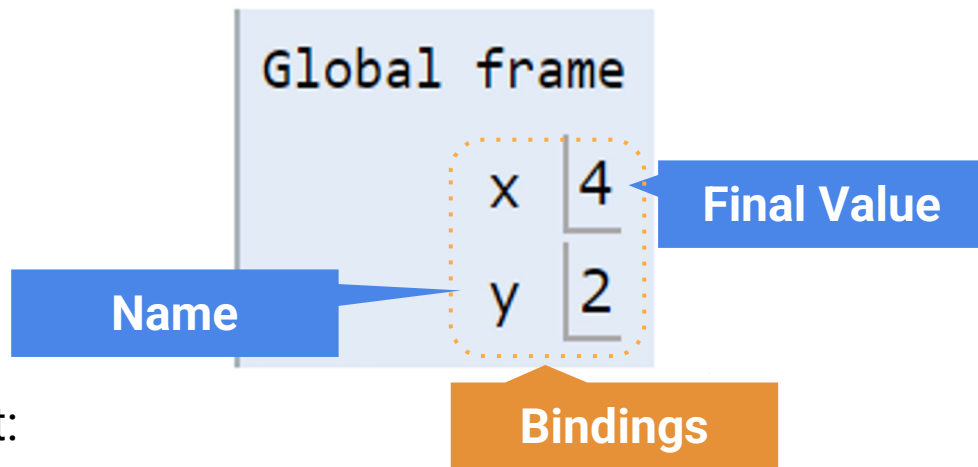
???

Visualizing Assignment

Demo

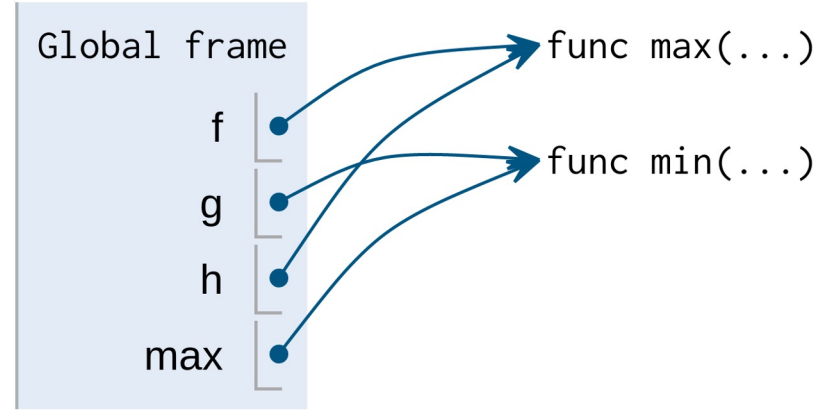
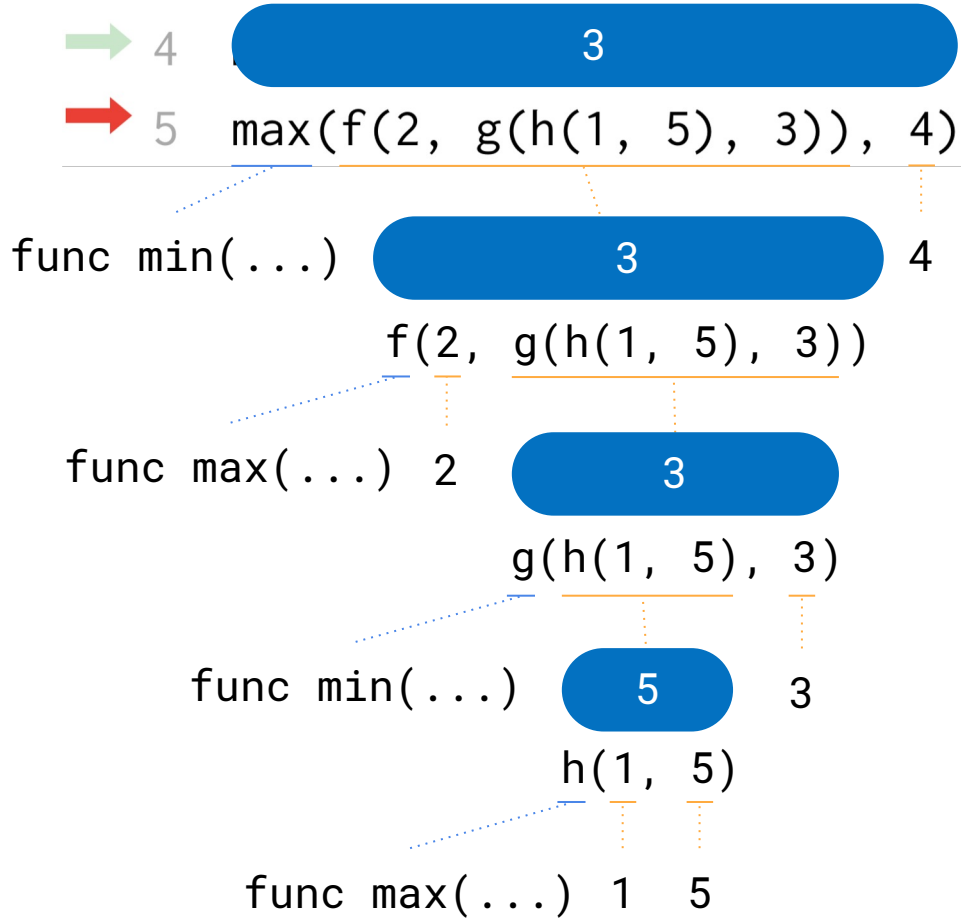
Names are bound to **values** in an **environment**

```
1  x = 1
2  y = 2
3  x = y * 2
```



To execute an assignment statement:

1. **Evaluate** the expression to the right of **=**.
2. **Bind** the value of the expression to the name to the left of **=** in the current environment.



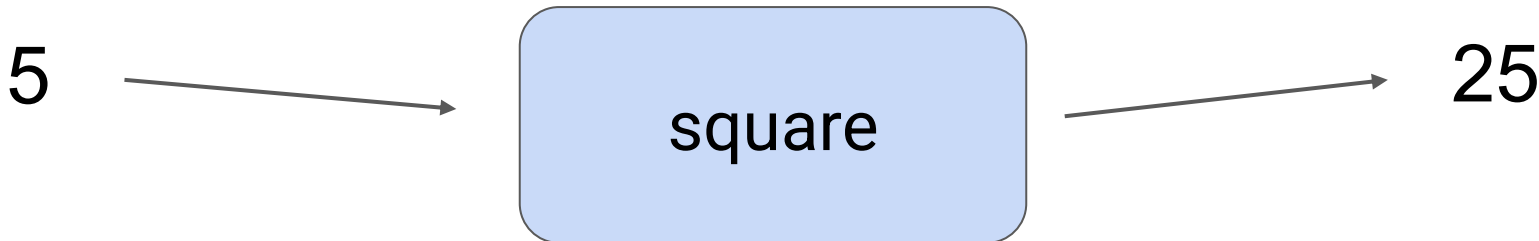
Functions

Functions

Functions allow us to abstract away entire expressions and 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

Demo

Function **signature** indicates name and number of arguments

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

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

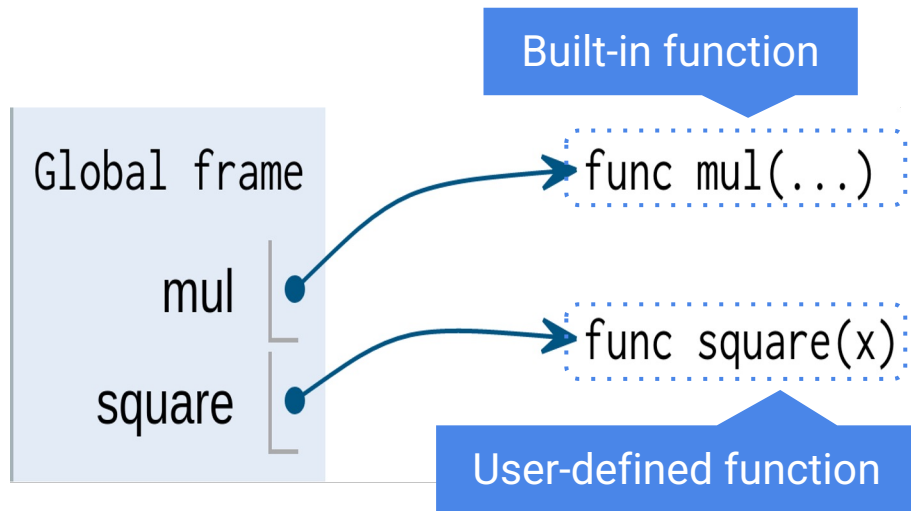
```
def square(x):  
    return x * x  
y = square(-2)
```

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

Functions in Environment Diagrams

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



def statements are a type of assignment that bind names to **function values**

Calling User-Defined Functions

Procedure for calling/applying user-defined functions (for now)

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

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

```
square(-2)
```

Global frame

square

func square(x)



Calling User-Defined Functions

Procedure for calling/applying user-defined functions (for now)

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

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

```
square(-2)
```

Local frame

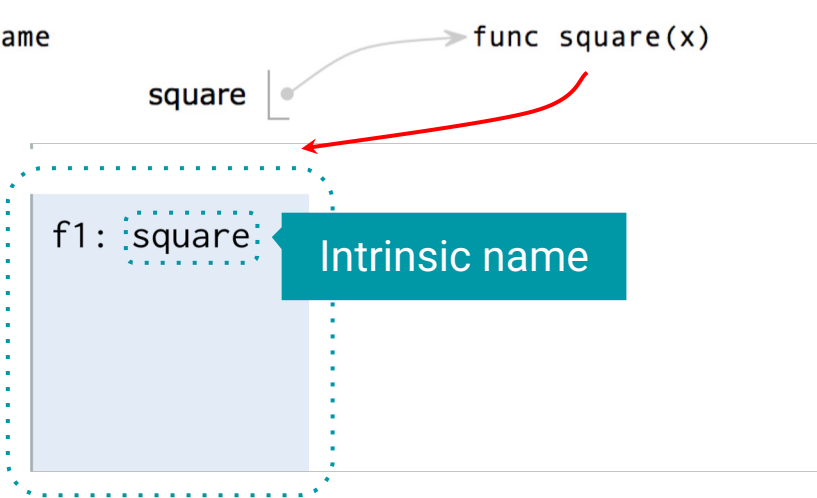
Global frame

square

func square(x)

f1: square

Intrinsic name



Calling User-Defined Functions

Procedure for calling/applying user-defined functions (for now)

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

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

```
square(-2)
```

Global frame

square

func square(x)

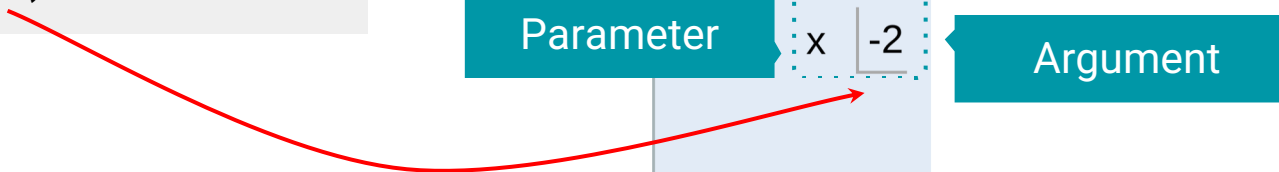
f1: square

Parameter

x

-2

Argument



Calling User-Defined Functions

Procedure for calling/applying user-defined functions (for now)

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

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

```
square(-2)
```

Global frame

square

func square(x)

f1: square

x

-2

Return
value

4

Putting it all together

1. Evaluate

- Evaluate the operator subexpression
- Evaluate each operand subexpression

2. Apply

- Apply the value of the operator subexpression to the values of the operand subexpression

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

Operator: square
Function: func square(x)

square:(1 - 3)

Operand: 1-3
Argument: -2

Local frame

Formal parameter
bound to argument

f1: square

x	-2
---	----

Return
value

4

Drawing Environment Diagrams

- Option 1: Python Tutor (tutor.cs61a.org)
 - Useful for quick visualization or for environment diagram questions
- Option 2: PythonAnywhere (editor.pythonanywhere.com)
 - Includes an integrated editor/interpreter
 - Good for more complicated code or if you want to debug
 - Developed by Rahul Arya

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.