

Lecture 1: Introduction

June 22nd, 2020



Welcome to CS 61A!

- 720 students
- 16 timezones



Humans of CS 61A

Instructors

Ryan Moughan

rmoughan@berkeley.edu



Chae Park

chae@berkeley.edu



Kavi Gupta

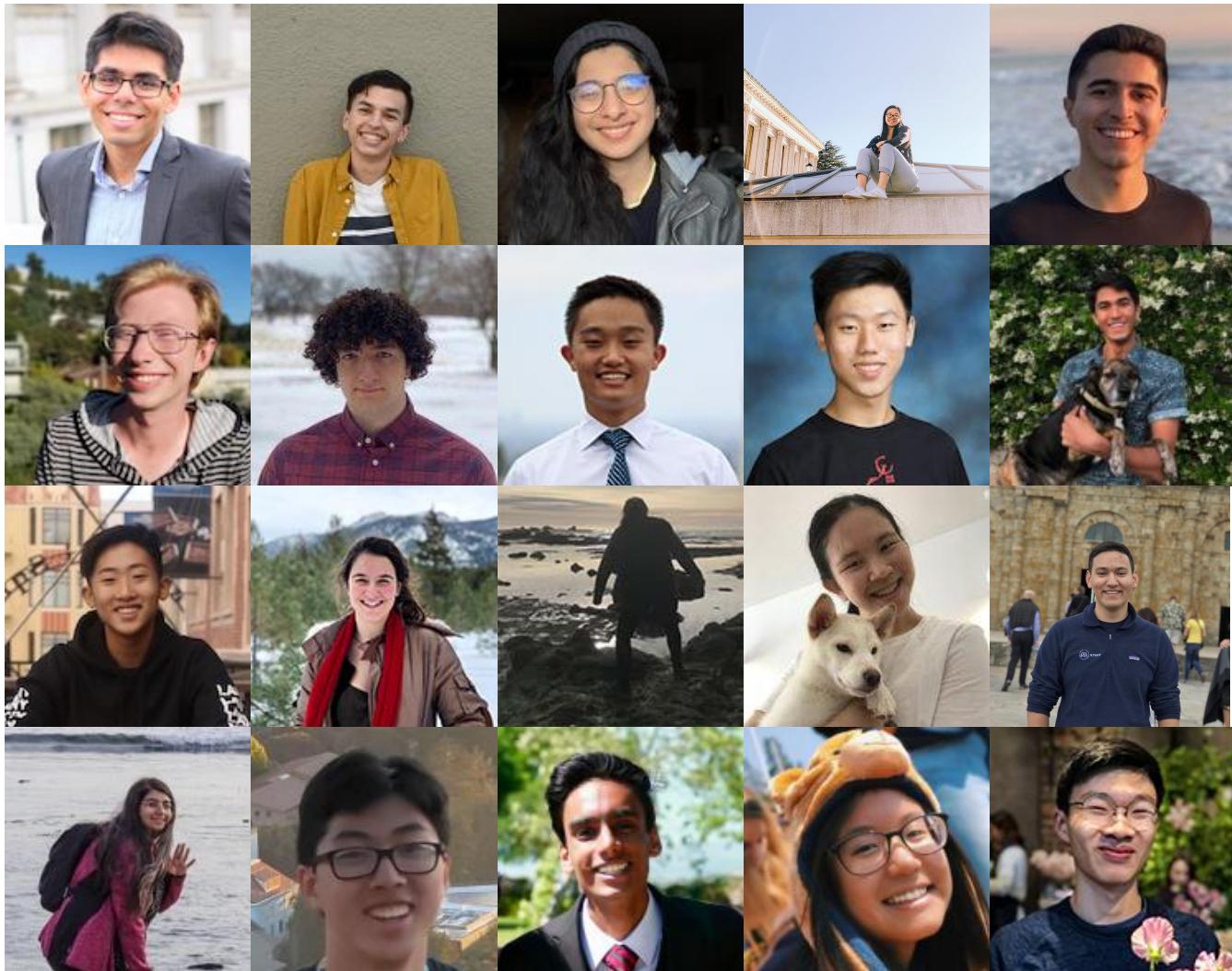
kavi@berkeley.edu



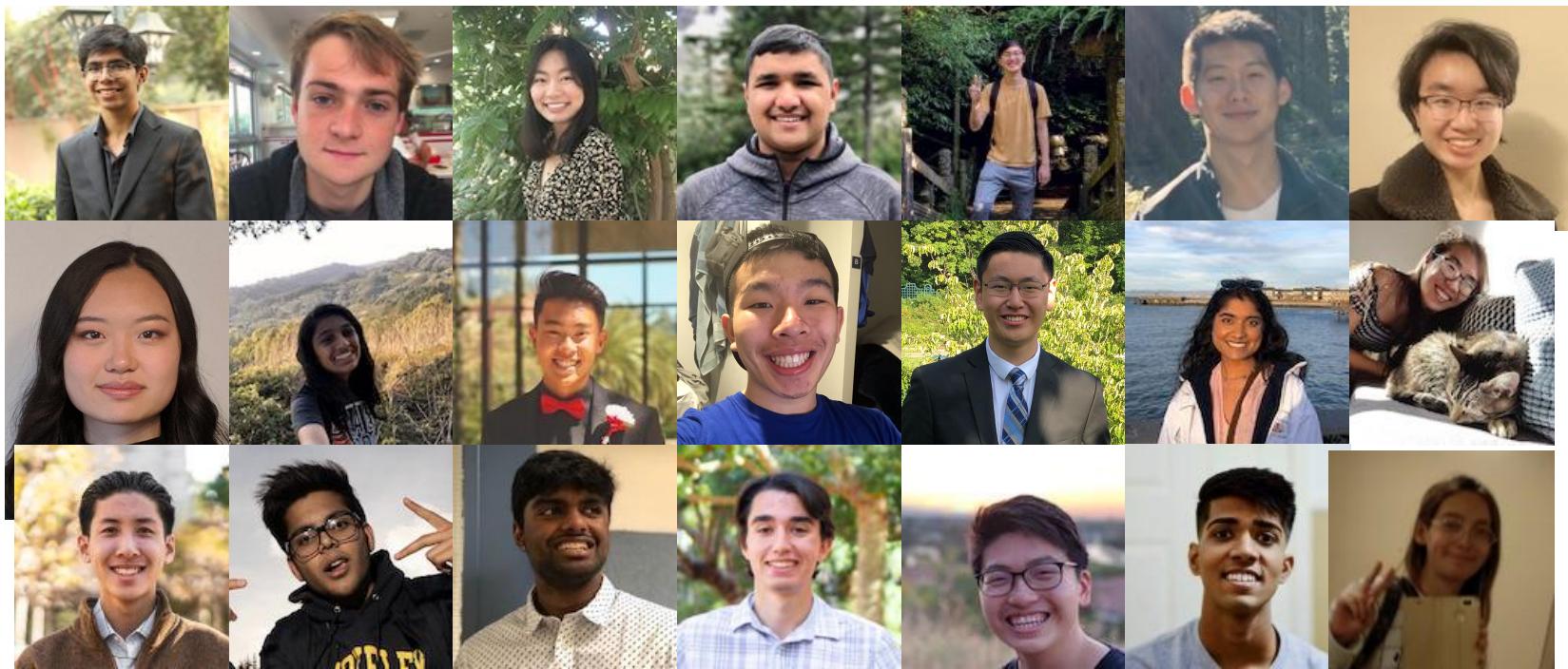
Prof. John DeNero



Teaching Assistants



Tutors



You!

Computer Science

What is Computer Science?

- What problems can be solved using computation?
- How do we solve those problems using computers?
- What techniques lead to effective solutions?

Systems

Artificial Intelligence

Graphics

Security

Networking

Programming Languages

Theory

...



What is CS 61A?

- A course about managing complexity
 - Mastering abstraction
 - Programming paradigms
- An introduction to programming
 - Full understanding of Python fundamentals
 - Combining multiple ideas in large projects
 - How computers interpret programming languages
- A challenging course that will demand a lot of you

Alternatives to CS 61A

CS 10 The Beauty and Joy of Computing

An introduction to fundamentals (& Python) that sets students up for success in CS 61A

cs10.org



Data 8 The Foundations of Data Science

Fundamentals of computing, statistical inference, & machine learning applied to real-world data sets

data8.org

Course Logistics

Course Format

Lecture	Whenever you want!
Lab	the most important part of this course
Discussion	the most important part of this course
Office hours	the most important part of this course
Tutoring	the most important part of this course
Textbook	composingprograms.com

- 8 programming **homeworks**
- 4 programming **projects**
- 1 **diagnostic quiz**, 1 **midterm exam**, and 1 **final exam**
- Lots of course support and a great community

The First Week

- Lab 0 released today!
- Discussion starts tomorrow!
- OH starts later this week

Lecture

- Main lectures are recorded videos by John Denero
- We will be giving supplementary live lectures once a week (details announced later)

Discussion Section

- Only part of the course that tracks attendance
- 90 minute section twice a week
- Largely worksheet based (but do not expect to finish it)
- Recorded

Office Hours

- Three formats: Appointments, Parties, and Instructor
- Appointment-based Office Hours:
 - 20 minutes each
 - Sign up the night before
 - Some will only be conceptual
- Parties:
 - 3 hours each
 - 2 flavors: Homework and Project
 - Queue-based
- Instructor:
 - Strong focus on concepts and not assignments

Small Group Tutoring Sections

Small-group sections (4-5 students) centered around a worksheet which reviews content from the corresponding discussion

Recurring

- Meet twice a week regularly with the same group of students
- Sign-ups will open later this week
- Start next week

Drop In

- Can sign up if you feel like you could use some extra reinforcement on topics presented from the last discussion
- Sign-ups will open every end of the week

Tools

Zoom: A platform for video calls

- Can ask questions via voice or text-chat
- Option to ask questions individually in a “breakout” section
- Where discussions , hw/project parties will happen

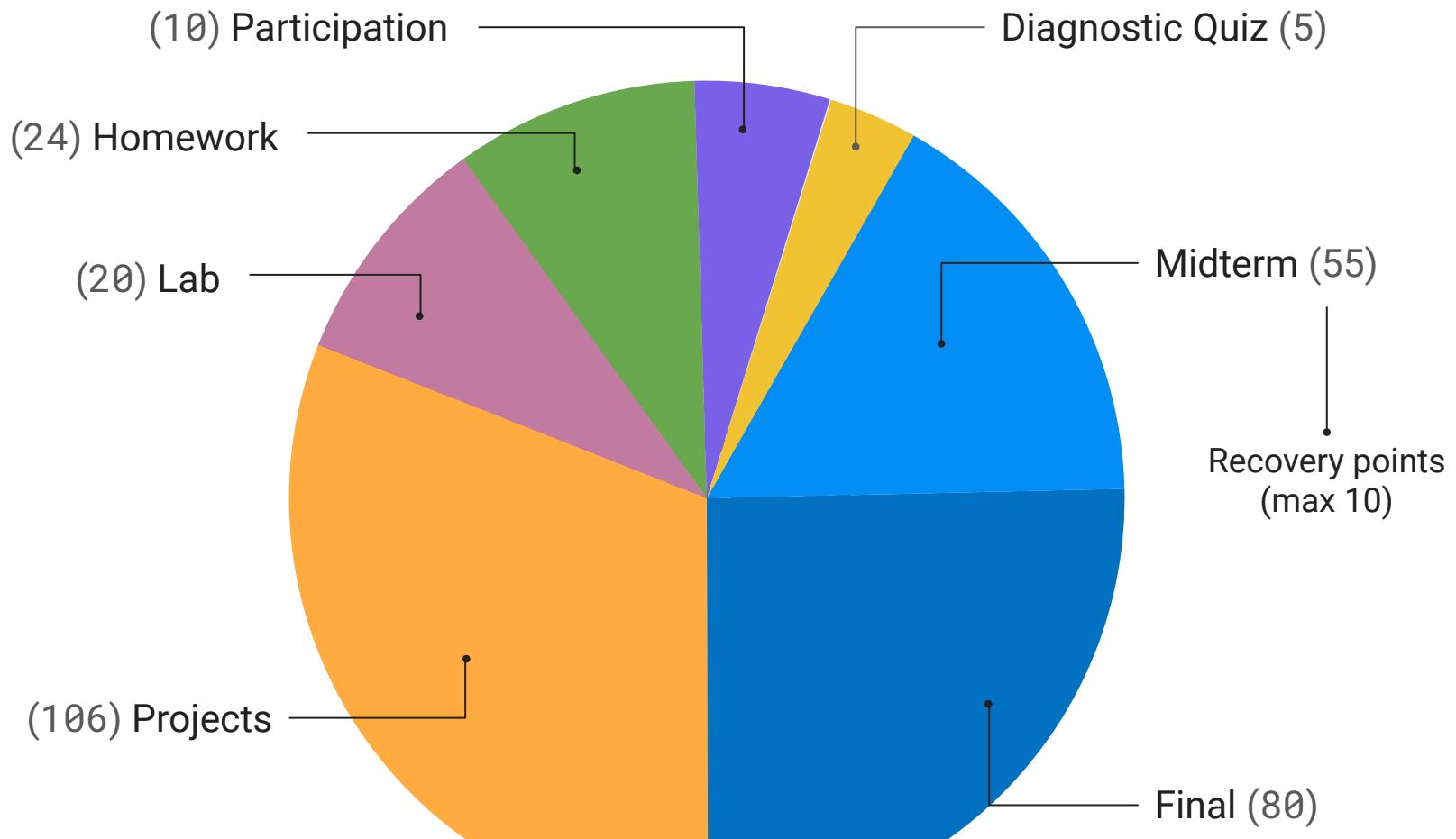
Piazza

- Forum for students to post questions & get announcements from instructors

Various pieces of software

- Will introduce most of these in lab00, so complete this ASAP!

Grading



Assignments

You can earn **150 points** through assignments:

- (10) 2 pt. Twice a week lab assignments
- (8) 3 pt. weekly programming homework assignments
- (4) 20-30 pt. programming projects

Most assignments are submitted using Ok (okpy.org).

You must have an @berkeley.edu address listed as your primary email on CalCentral to be enrolled on Ok.

If you have not been added to OK—which you'll find out when you do your assignments—fill out the form on Piazza / email your TA ASAP

Lab Assignments

- Generally released Mon/Wed, due Wed/Fri respectively
- Graded on correct completion, all or nothing
- Lowest two lab grades dropped
- You will complete these on your own time - however, you may find that it is helpful to work on labs during office hours to get support
- More details: <https://cs61a.org/articles/about.html#labs>

Homework Assignments

- Generally released on Wed, due next Tues
- Graded on correctness, partial credit with every incorrect answer losing you one point on the homework (up till 0)
- **Homework Recovery:**
 - Can recovery one incorrect question per homework by going through one of hw recovery processes
 - Homework recovery session
 - Appointment based office hour
- More details: <https://cs61a.org/articles/about.html#homework>

Participation

Discussion Participation

- This is part of 300 points.
- You can earn up to **10 participation points**.

Class Participation

- This is **not** part of 300 points.
- Can be used for exam recovery
- Each of the following opportunities is worth 1 class participation credit:
 - Weekly student survey (~8 possible)
 - Extra discussion section attendance (after the initial 10, 2 possible)
 - Extra lab assignment submission (after the initial 10, 2 possible)

EPA (Efforts, Participation, Altruism)

- Extra credit(s); not part of 300 points.
- Can be earned through (but not limited to):
 - Effort = {Office hours, doing every single lab, hw, reading Piazza pages, etc.}
 - Participation = {Raising hand in discussion, asking Piazza questions, etc.}
 - Altruism = {Helping other students, answering Piazza or Office Hour questions etc.}
- Scoring will remain confidential.

Exams

Diagnostic Quiz (5 pts)

When: Thursday, July 2 @ TBA

Format: 60-90 minute electronic exam

Midterm (55 pts)

When: Thursday, July 16 @ TBA

Format: 180 minute electronic exam

Final exam (80 pts)

When: Thursday, August 13 @ TBA

Format: 180 minute electronic exam

Exam recovery

After getting >5 of class participation points, 10 credits can be used toward exam recovery points for the Midterm

Alternates

We will have alternates for each exam at a 12h offset. Fill out

<http://links.cs61a.org/alt>

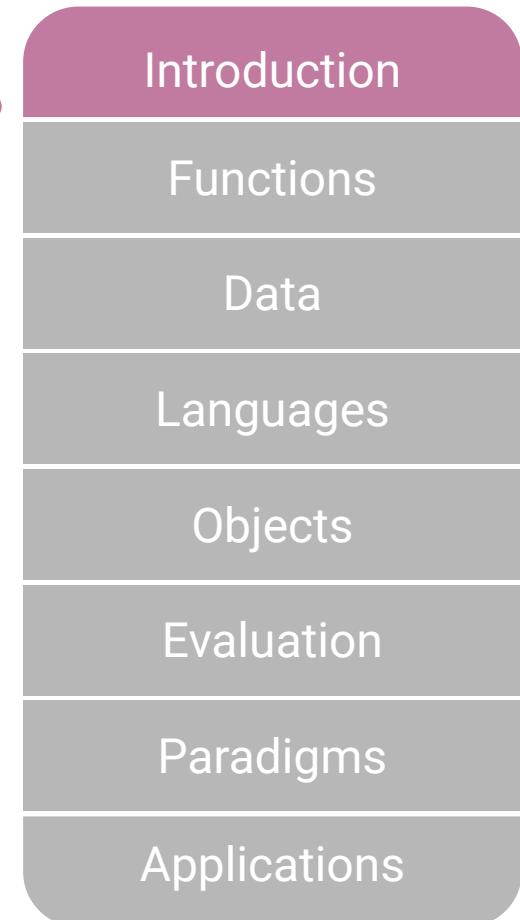
Collaboration

- This course is not curved -- collaboration, not competition, is key
- Asking questions and discussing ideas is highly encouraged
- **The only students with whom you can share code are**
 - **Your project partner**
 - **Students who have finished the problem you are working on**
- More info: <https://cs61a.org/articles/about.html#academic-honesty>

Course Overview

Every week will center around a theme with a specific set of goals.

- Learn the fundamentals of programming
- Become comfortable with Python



Expressions

What's in a program?

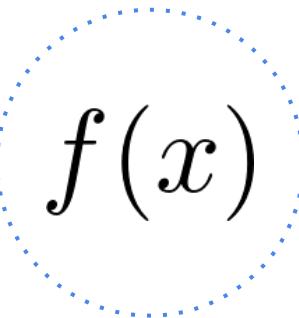
- Programs work by manipulating **values**
- **Expressions** in programs evaluate to values
 - **Primitive expressions** evaluate directly to values with minimal work needed
- **Operators** combine primitives expressions into more complex expressions
- The Python interpreter evaluates expressions and displays their values

$$20 + 17$$

$$2^{100}$$

$$\sin \pi$$

$$\lim_{x \rightarrow \infty} \frac{1}{x}$$


$$f(x)$$

$$\frac{20}{17}$$

$$\sum_{i=1}^n i$$

$$\sqrt{2017}$$

$$\binom{n}{x}$$

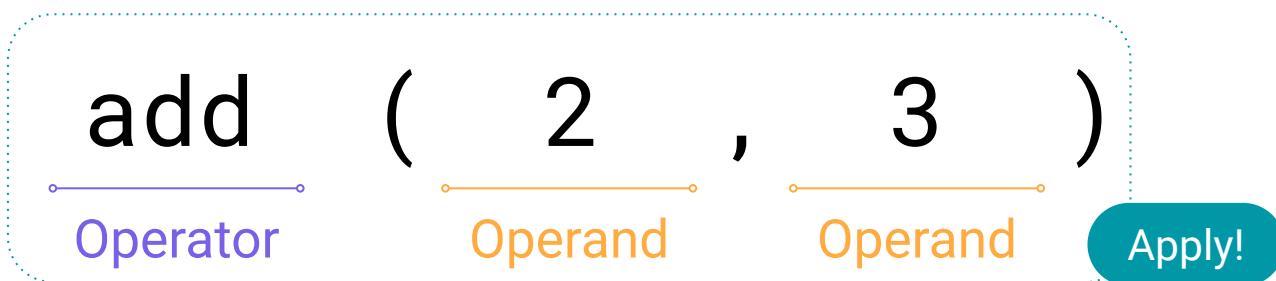
$$| - 2017 |$$

An **expression** describes a computation and evaluates to a value.

Call Expressions

Evaluation procedure for **call expressions**

1. Evaluate the **operator**
2. Evaluate the **operands** from left to right
3. **Apply** the operator (a **function**) to the evaluated operands (**arguments**)



Operators and operands are also expressions

So they also *evaluate to values*

add(add(6, mul(4, 6)), mul(3, 5))
???

Nested Call Expressions

- Humans evaluate inside-out

```
add(add(6, mul(4, 6)), mul(3, 5))  
add(add(6, 24 ), mul(3, 5))  
add(add(6, 24 ), mul(3, 5))  
add( 30 , mul(3, 5))  
add( 30 , mul(3, 5))  
add( 30 , 15 )  
add( 30 , 15 )  
45
```

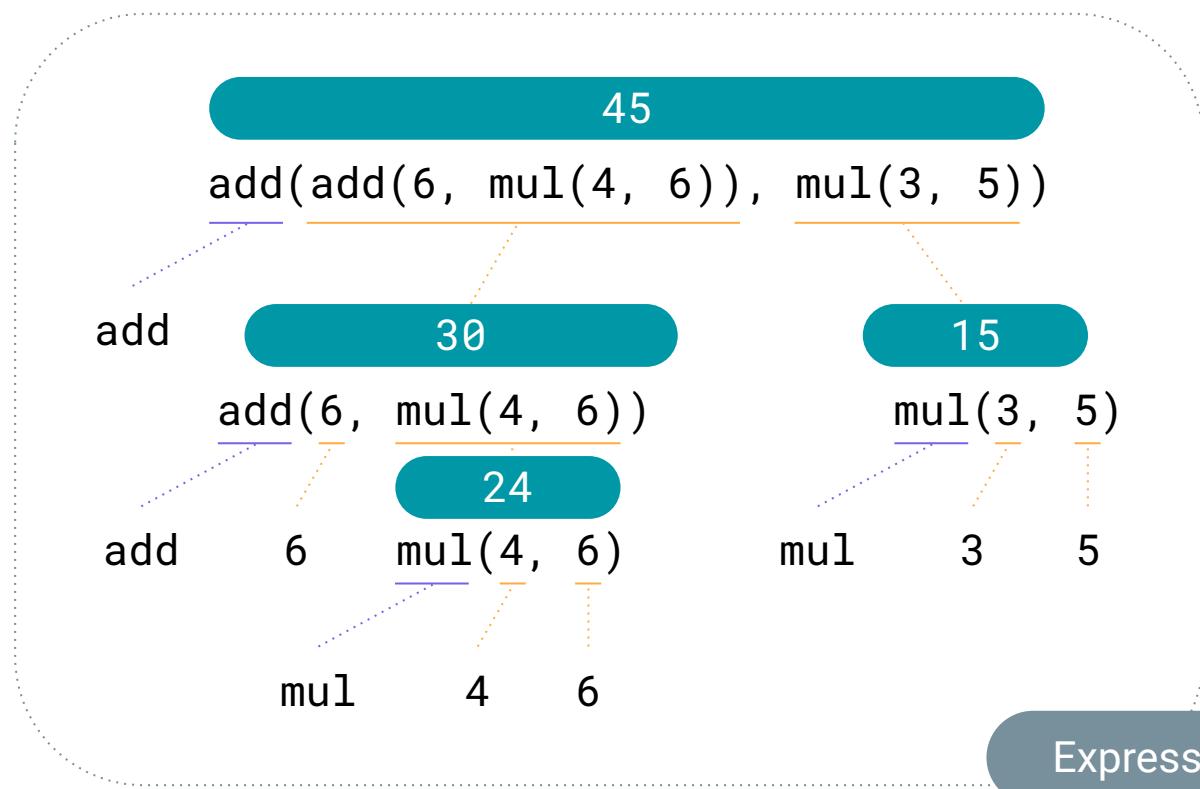
- We can jump ahead or skip around, but Python can't do that!
- How does the computer know which call to evaluate first?

Nested Call Expressions

1 Evaluate operator

2 Evaluate operands

3 Apply!



Demo

Functions, Objects, & Interpreters

61A Lecture 2

Announcements

Names, Assignment, and User-Defined Functions

(Demo)

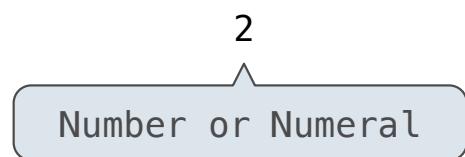
Types of Expressions

Types of Expressions

Primitive expressions:

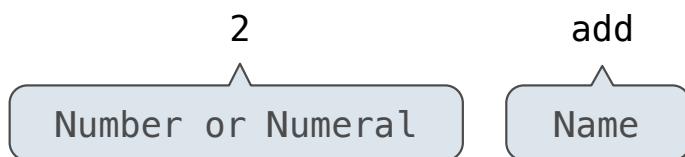
Types of Expressions

Primitive expressions:



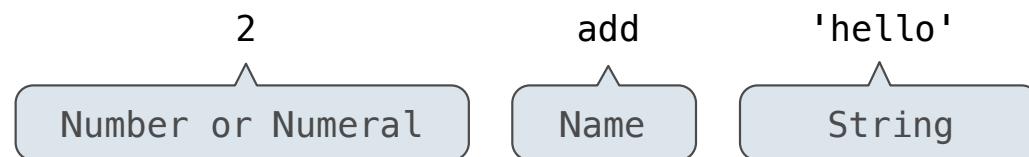
Types of Expressions

Primitive expressions:



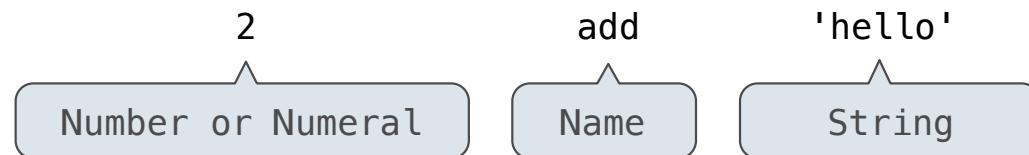
Types of Expressions

Primitive expressions:



Types of Expressions

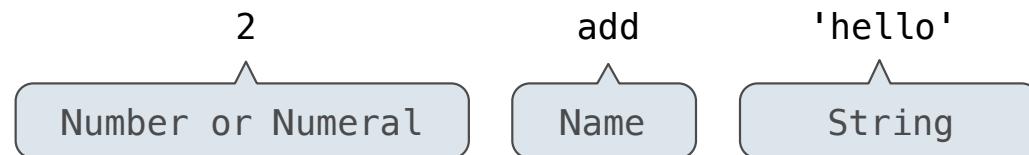
Primitive expressions:



Call expressions:

Types of Expressions

Primitive expressions:

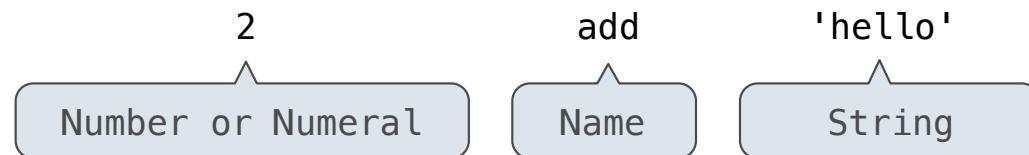


Call expressions:

max (2 , 3)

Types of Expressions

Primitive expressions:

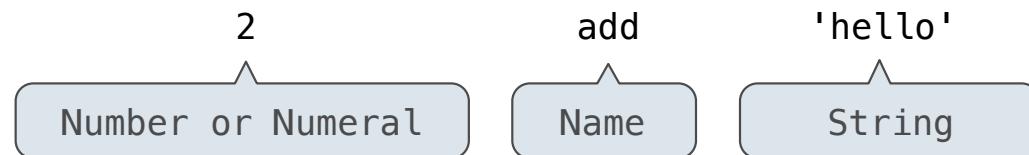


Call expressions:

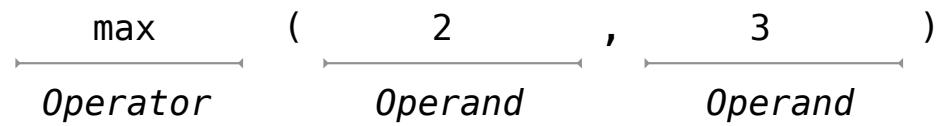


Types of Expressions

Primitive expressions:

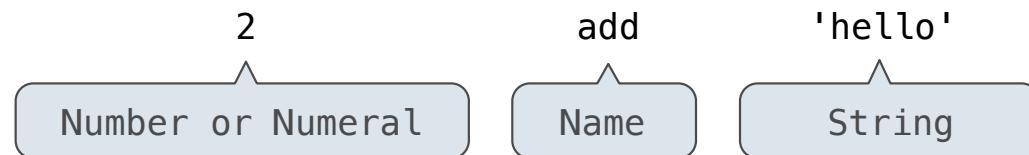


Call expressions:

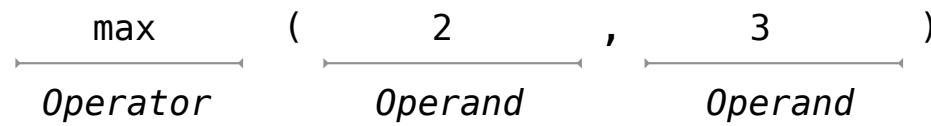


Types of Expressions

Primitive expressions:



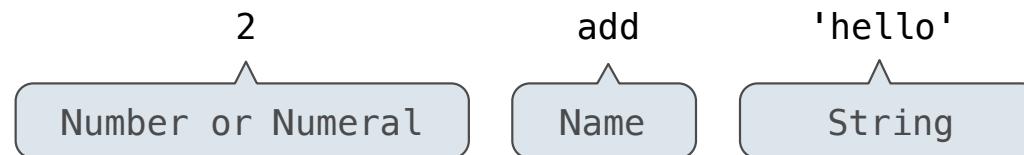
Call expressions:



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

Types of Expressions

Primitive expressions:



Call expressions:

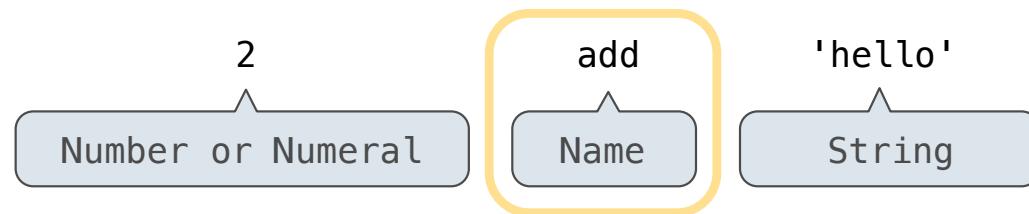


An operand can also
be a call expression

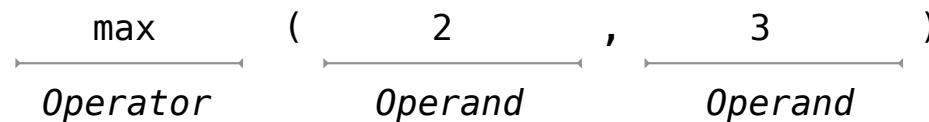
max(min(pow(3, 5), -4), min(1, -2))

Types of Expressions

Primitive expressions:



Call expressions:



An operand can also be a call expression

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

Discussion Question 1

Discussion Question 1

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

Discussion Question 1

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

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

Discussion Question 1

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

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

Discussion Question 1

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

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

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

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

Discussion Question 1

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

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

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

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

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

Discussion Question 1

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

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

Discussion Question 1

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

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

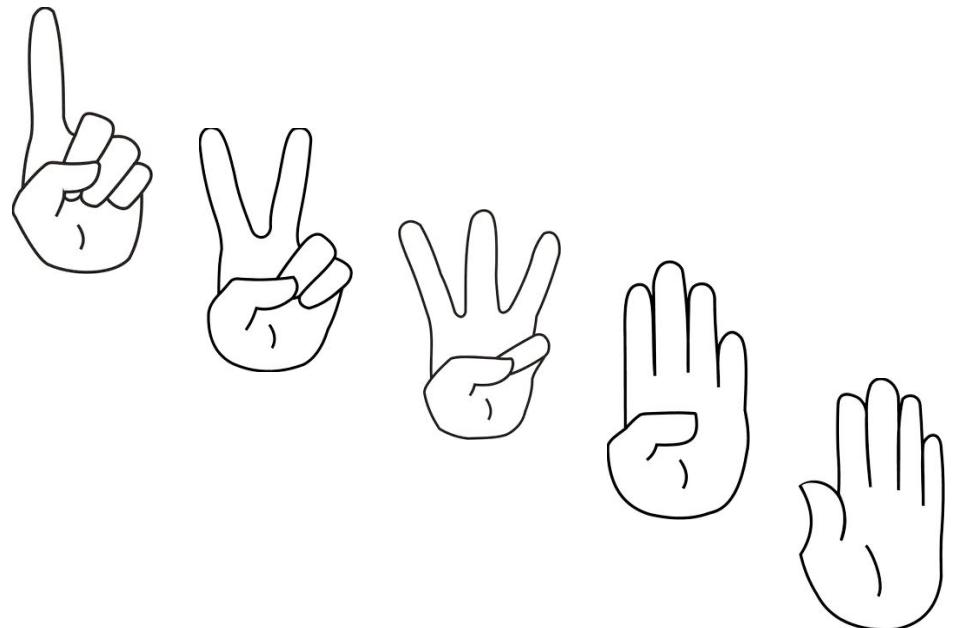
???

Discussion Question 1

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

```
>>> 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.

[Interactive Diagram](#)

Environment Diagrams

Environment diagrams visualize the interpreter's process.

```
→ 1 from math import pi
→ 2 tau = 2 * pi
```

[Interactive Diagram](#)

Environment Diagrams

Environment diagrams visualize the interpreter's process.

```
→ 1 from math import pi  
→ 2 tau = 2 * pi
```

Global frame
pi | 3.1416

[Interactive Diagram](#)

Environment Diagrams

Environment diagrams visualize the interpreter's process.

→ 1 from math import pi
→ 2 tau = 2 * pi

Global frame
pi | 3.1416

Code (left):

Frames (right):

[Interactive Diagram](#)

Environment Diagrams

Environment diagrams visualize the interpreter's process.

```
→ 1 from math import pi  
→ 2 tau = 2 * pi
```

Global frame
pi | 3.1416

Code (left):

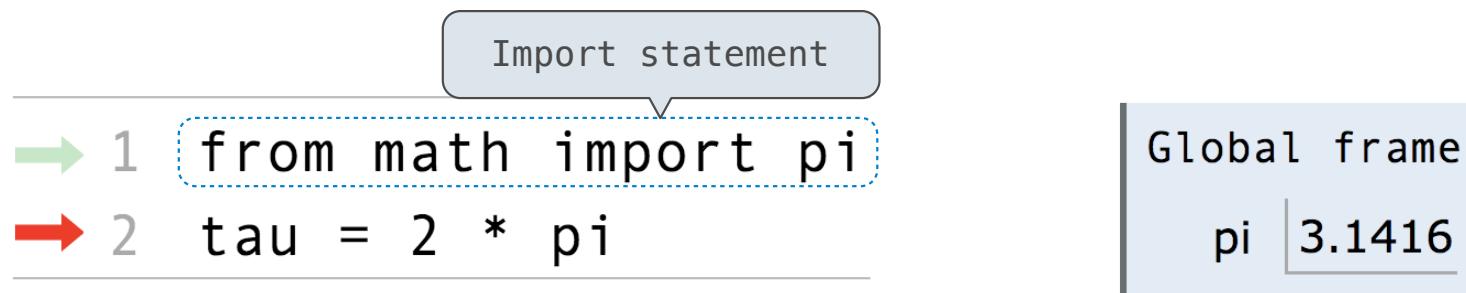
Statements and expressions

Frames (right):

[Interactive Diagram](#)

Environment Diagrams

Environment diagrams visualize the interpreter's process.



Code (left):

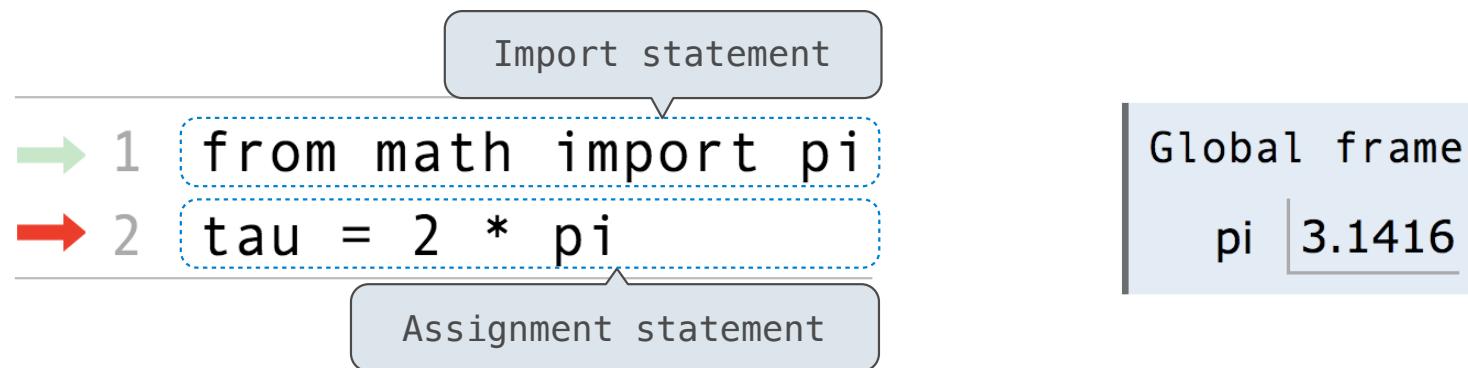
Statements and expressions

Frames (right):

[Interactive Diagram](#)

Environment Diagrams

Environment diagrams visualize the interpreter's process.



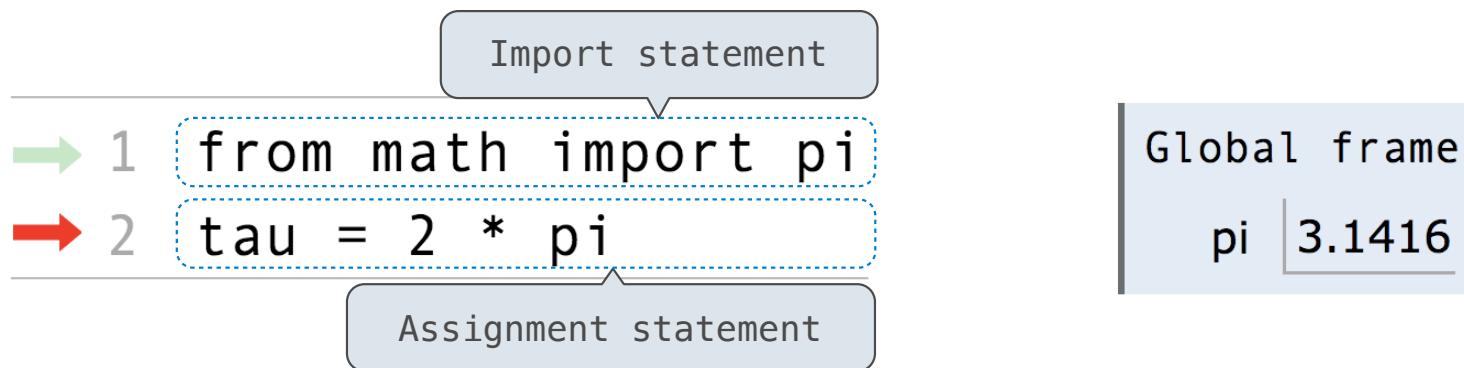
Code (left):
Statements and expressions

Frames (right):

[Interactive Diagram](#)

Environment Diagrams

Environment diagrams visualize the interpreter's process.



Code (left):

Statements and expressions

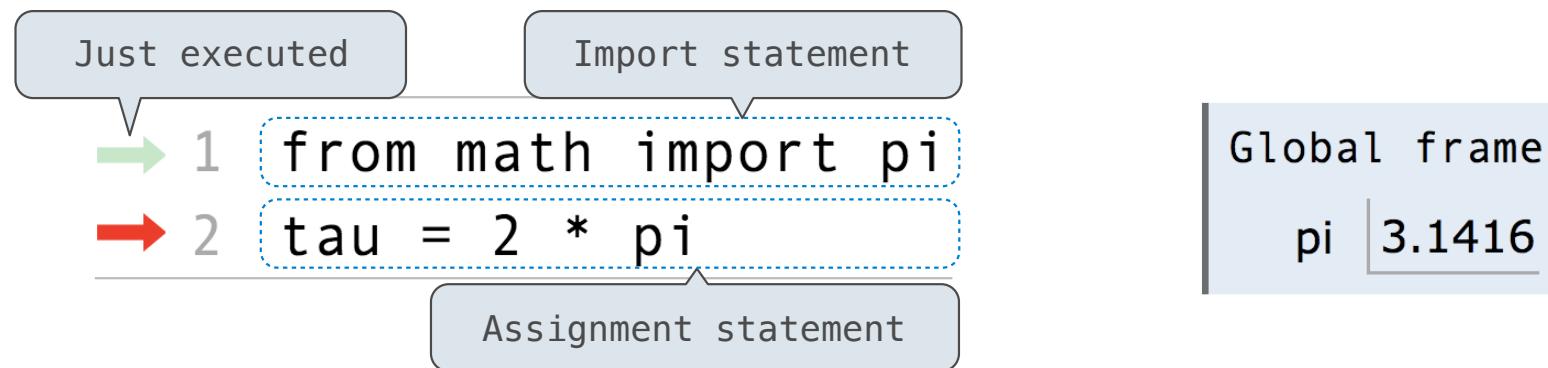
Frames (right):

Arrows indicate evaluation order

[Interactive Diagram](#)

Environment Diagrams

Environment diagrams visualize the interpreter's process.



Code (left):

Statements and expressions

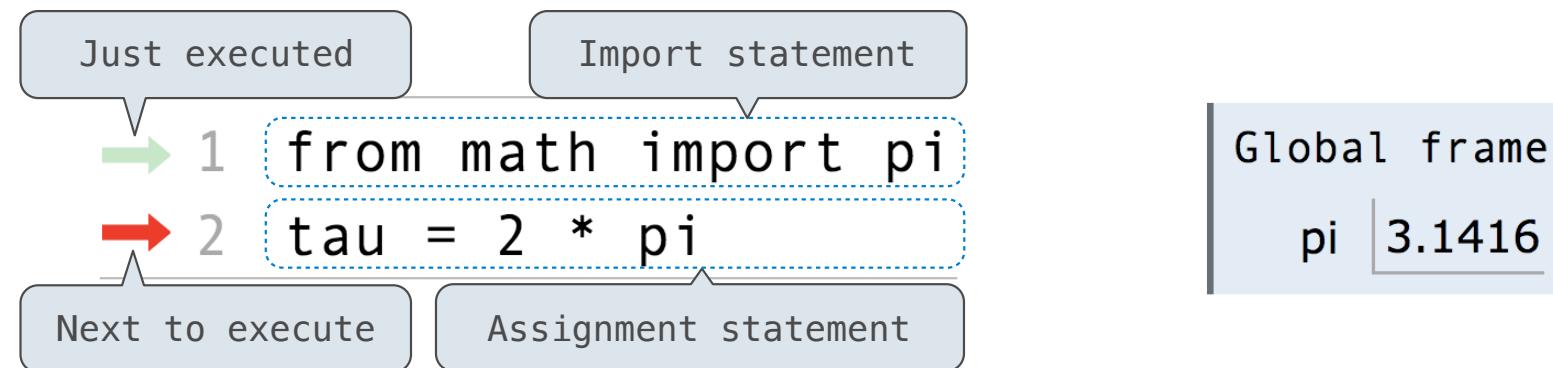
Frames (right):

Arrows indicate evaluation order

[Interactive Diagram](#)

Environment Diagrams

Environment diagrams visualize the interpreter's process.



Code (left):

Statements and expressions

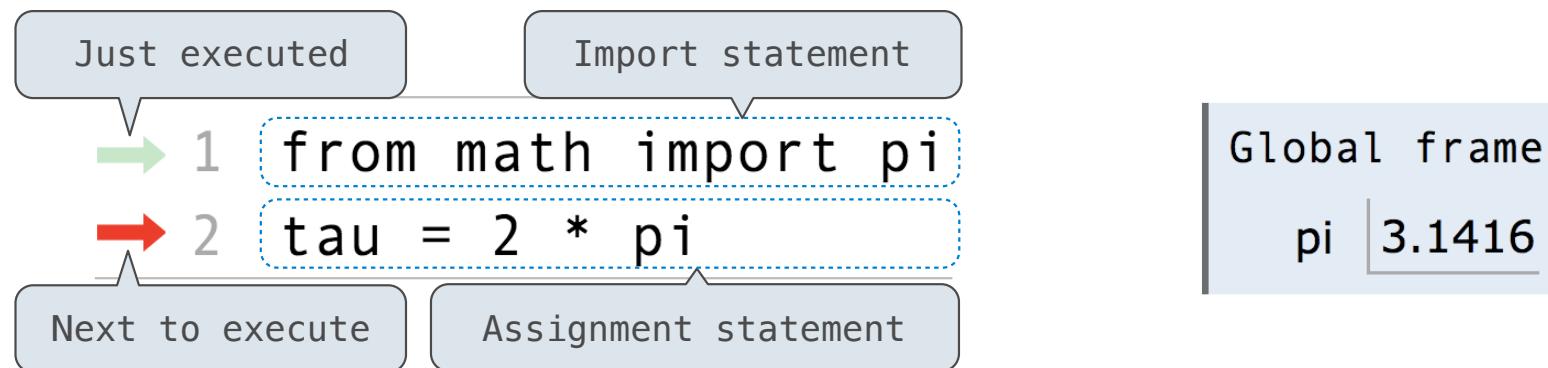
Frames (right):

Arrows indicate evaluation order

[Interactive Diagram](#)

Environment Diagrams

Environment diagrams visualize the interpreter's process.



Code (left):

Statements and expressions

Arrows indicate evaluation order

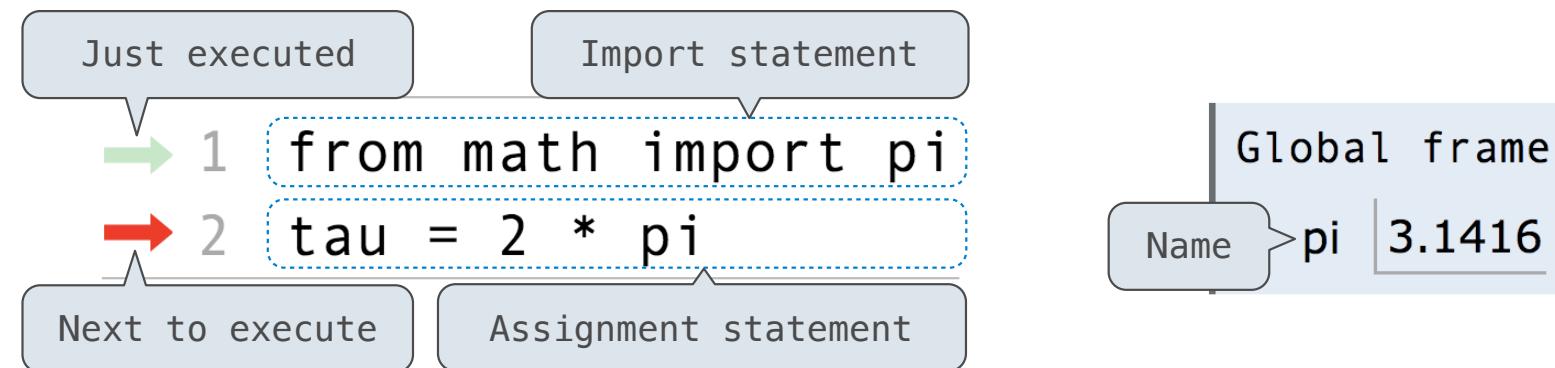
Frames (right):

Each name is bound to a value

[Interactive Diagram](#)

Environment Diagrams

Environment diagrams visualize the interpreter's process.



Code (left):

Statements and expressions

Arrows indicate evaluation order

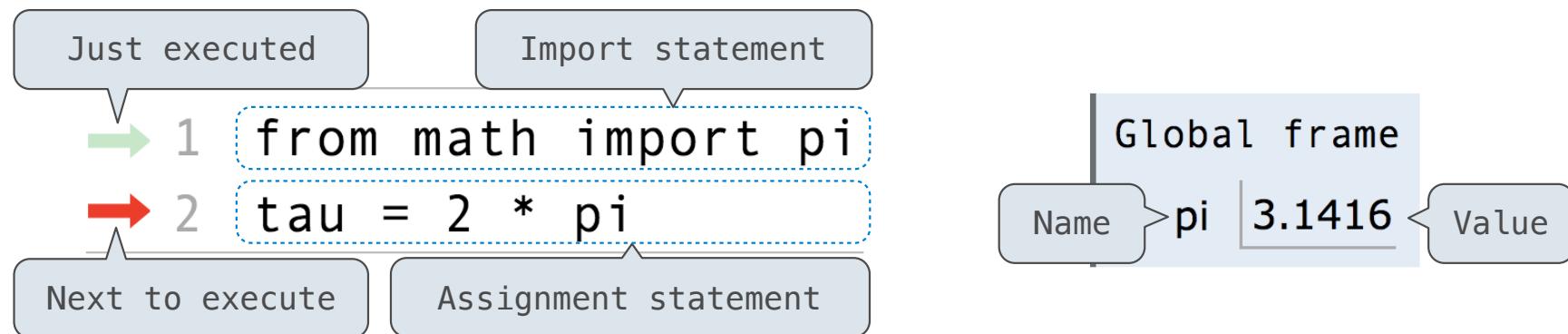
Frames (right):

Each name is bound to a value

[Interactive Diagram](#)

Environment Diagrams

Environment diagrams visualize the interpreter's process.



Code (left):

Statements and expressions

Arrows indicate evaluation order

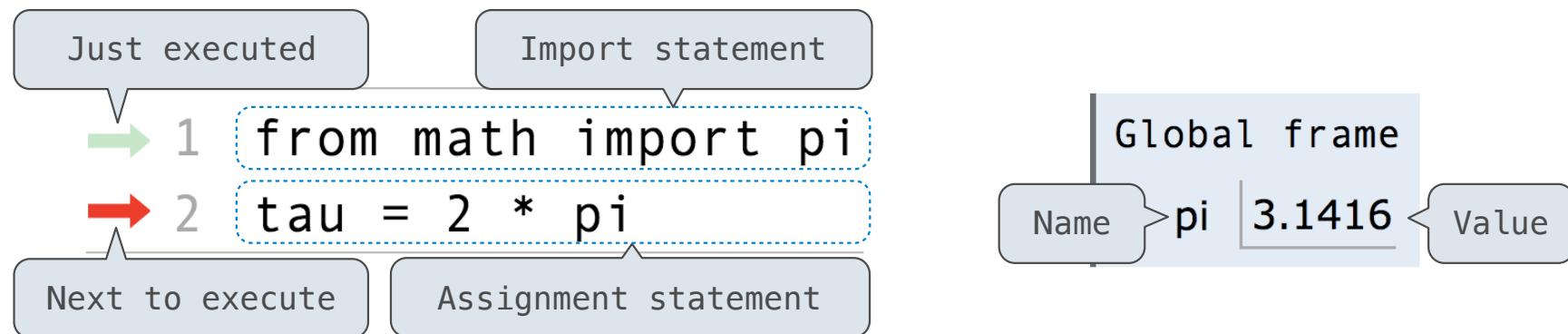
Frames (right):

Each name is bound to a value

[Interactive Diagram](#)

Environment Diagrams

Environment diagrams visualize the interpreter's process.



Code (left):

Statements and expressions

Arrows indicate evaluation order

Frames (right):

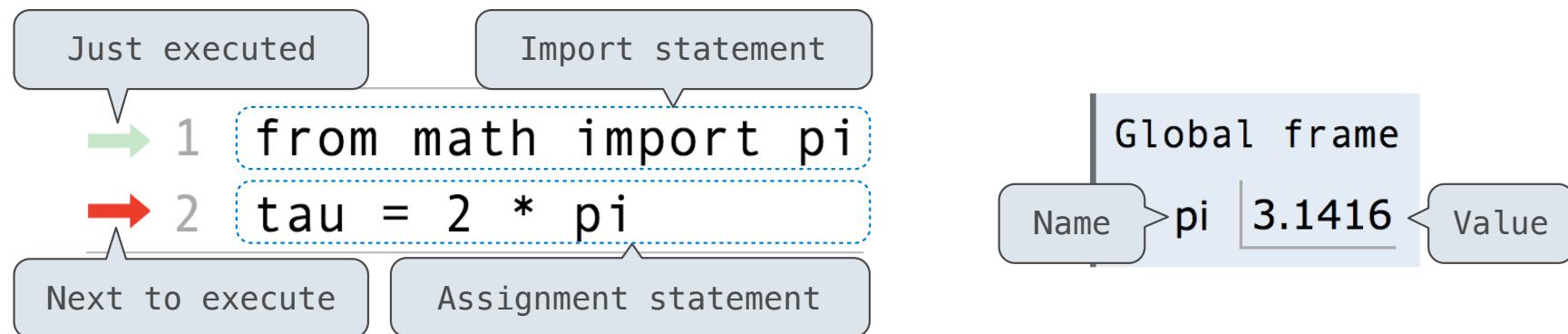
Each name is bound to a value

Within a frame, a name cannot be repeated

[Interactive Diagram](#)

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)

[Interactive Diagram](#)

Assignment Statements

[Interactive Diagram](#)

8

Assignment Statements

```
1 a = 1
→ 2 b = 2
→ 3 b, a = a + b, b
```

[Interactive Diagram](#)

Assignment Statements

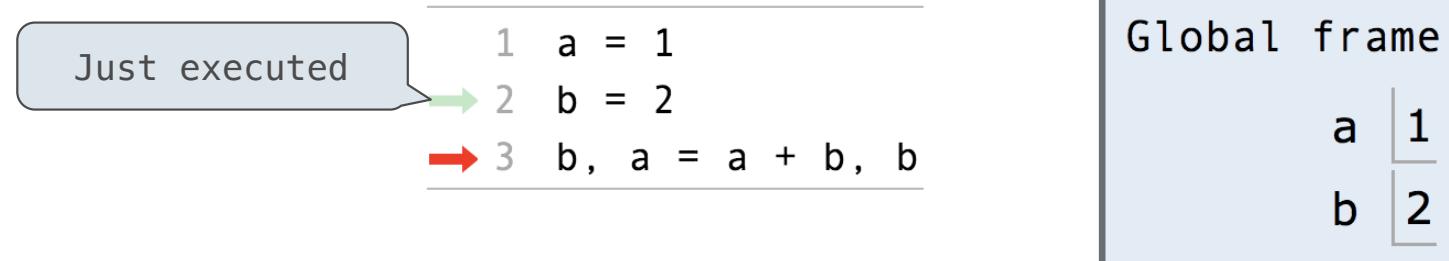
```
1 a = 1
→ 2 b = 2
→ 3 b, a = a + b, b
```

Global frame

a	1
b	2

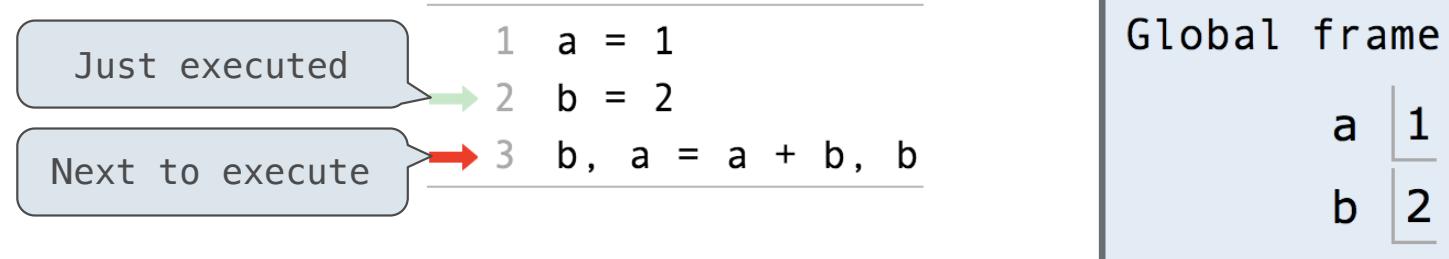
[Interactive Diagram](#)

Assignment Statements



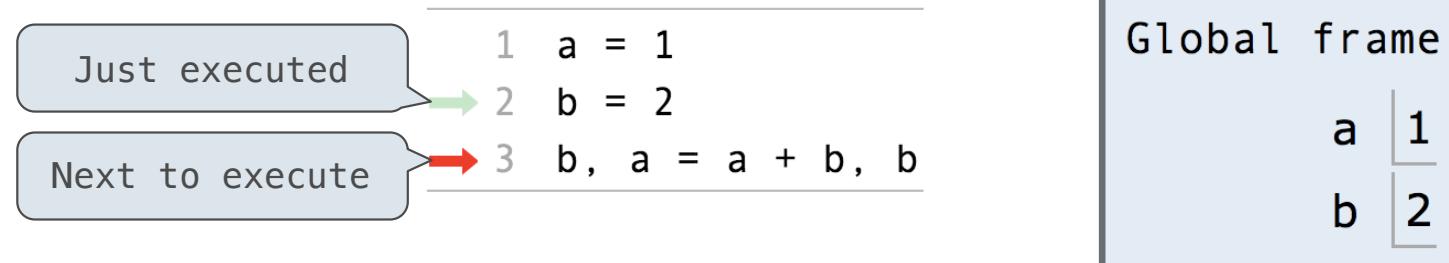
[Interactive Diagram](#)

Assignment Statements



[Interactive Diagram](#)

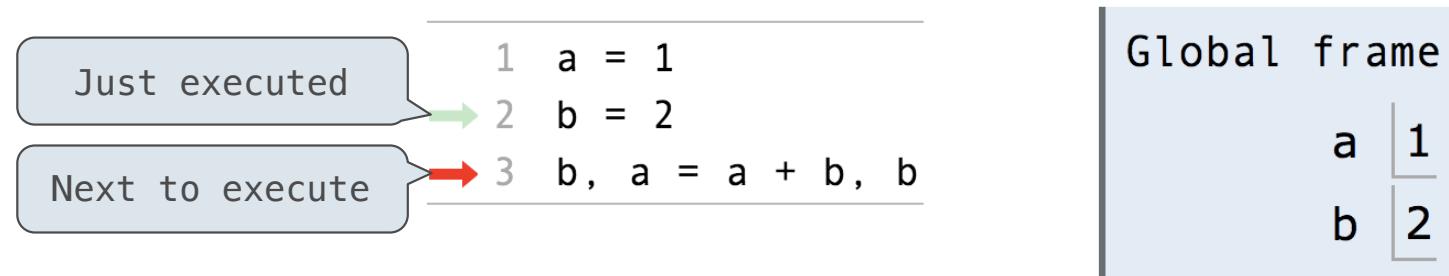
Assignment Statements



Execution rule for assignment statements:

[Interactive Diagram](#)

Assignment Statements

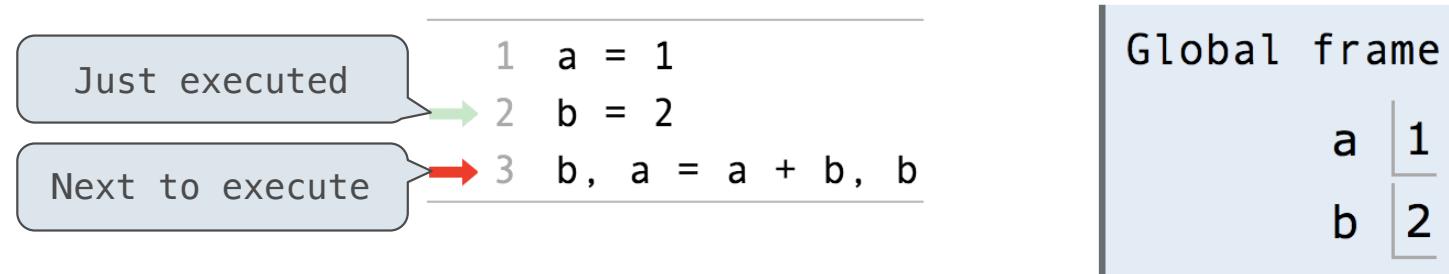


Execution rule for assignment statements:

1. Evaluate all expressions to the right of `=` from left to right.

[Interactive Diagram](#)

Assignment Statements

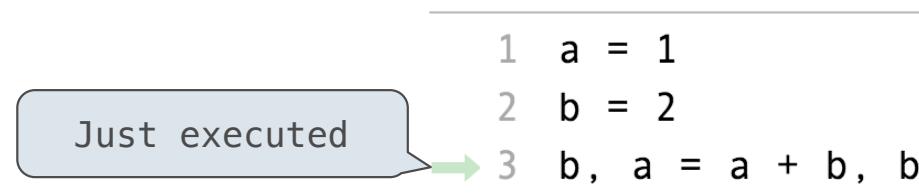
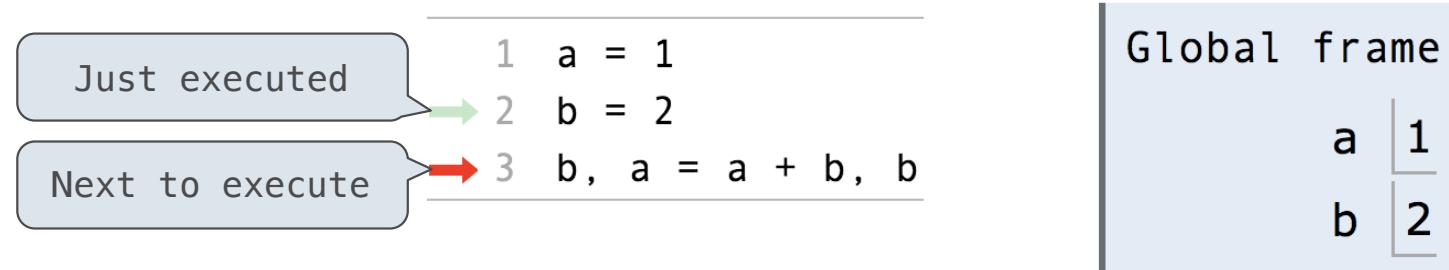


Execution rule for assignment statements:

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.

[Interactive Diagram](#)

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.

[Interactive Diagram](#)

Assignment Statements

Just executed 1 a = 1
Next to execute 2 b = 2
 3 b, a = a + b, b

Global frame

a	1
b	2

Just executed 1 a = 1
 2 b = 2
 3 b, a = a + b, b

Global frame

a	2
b	3

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.

[Interactive Diagram](#)

Discussion Question 1 Solution

(Demo)

[Interactive Diagram](#)

Discussion Question 1 Solution

(Demo)

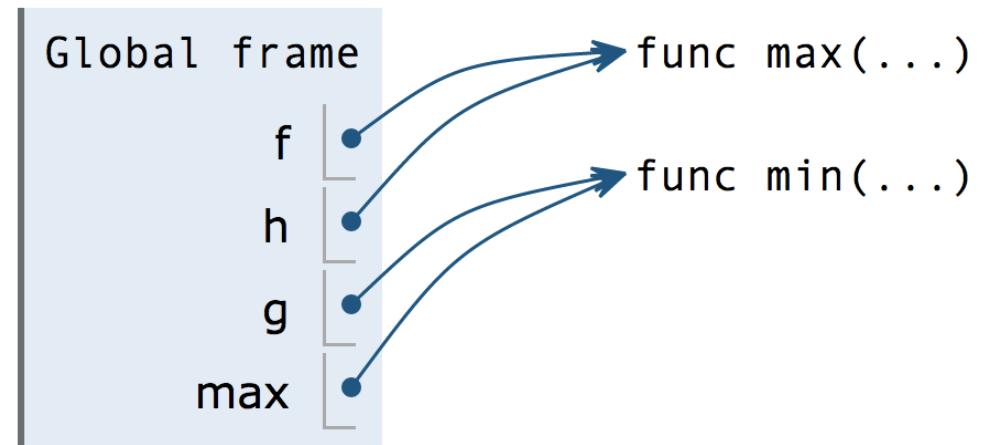
-
- 1 f = min
 - 2 f = max
 - 3 g, h = min, max
 - 4 max = g
 - 5 max(f(2, g(h(1, 5), 3)), 4)

[Interactive Diagram](#)

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)
```

(Demo)

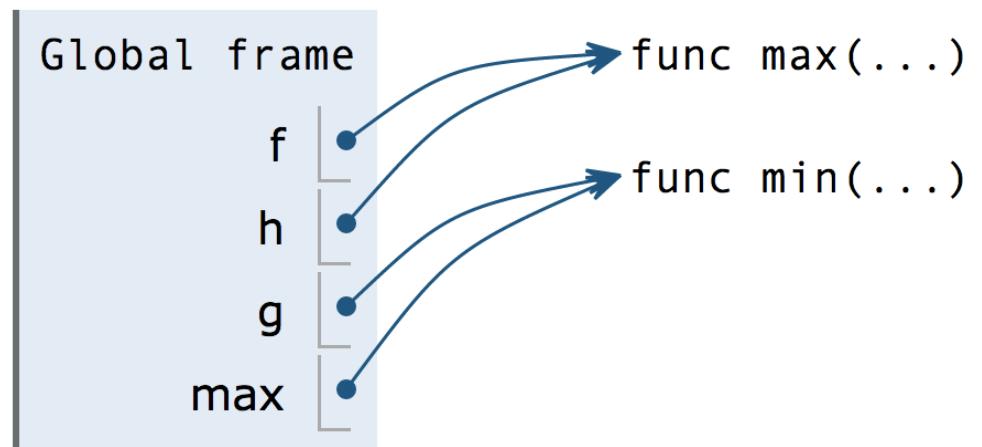


[Interactive Diagram](#)

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)
```

(Demo)

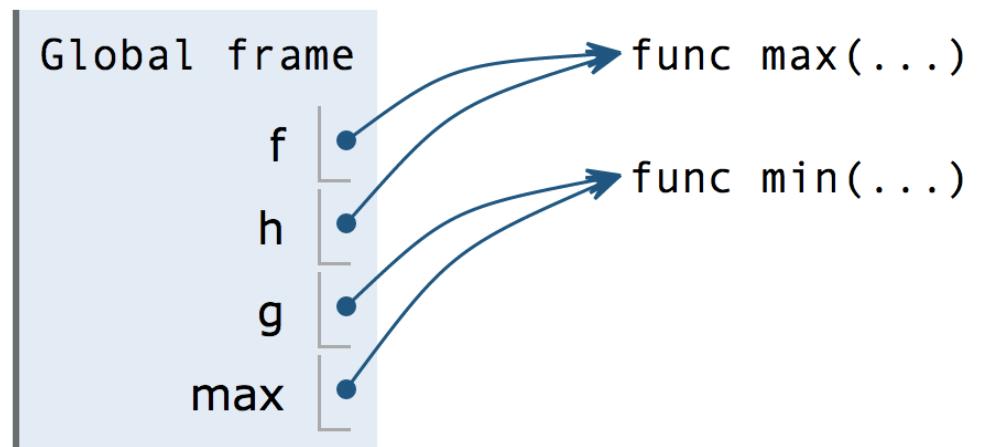


[Interactive Diagram](#)

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)  
  
func min(...)
```

(Demo)



[Interactive Diagram](#)

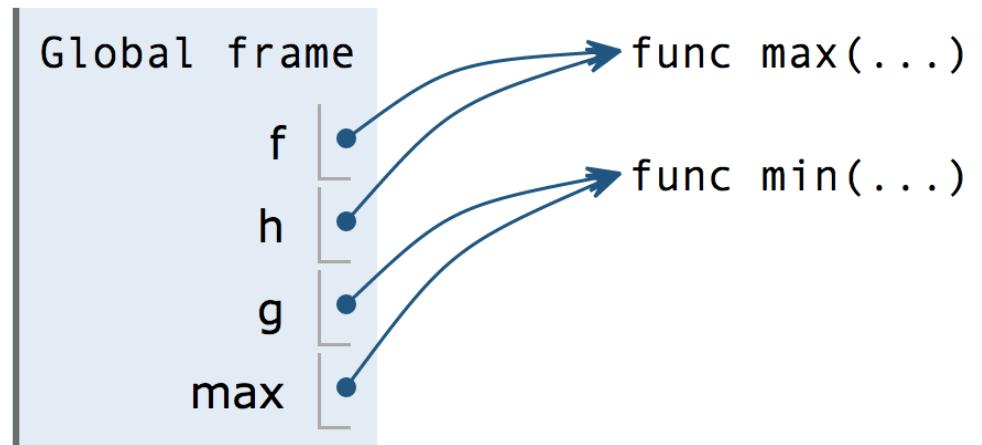
Discussion Question 1 Solution

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

func min(...)

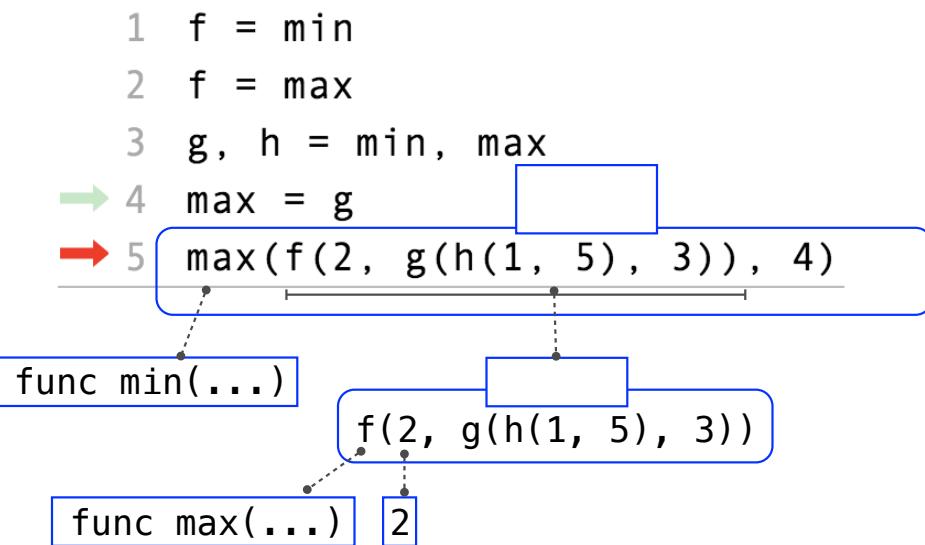
f(2, g(h(1, 5), 3))

(Demo)

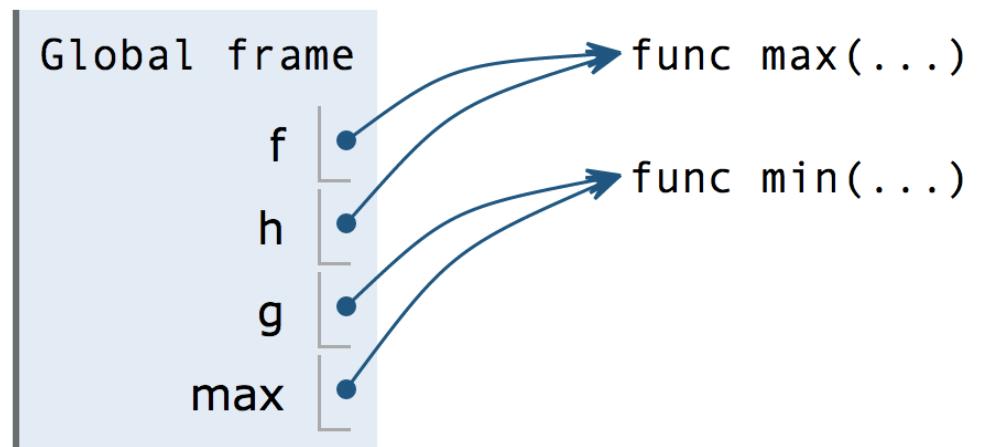


[Interactive Diagram](#)

Discussion Question 1 Solution



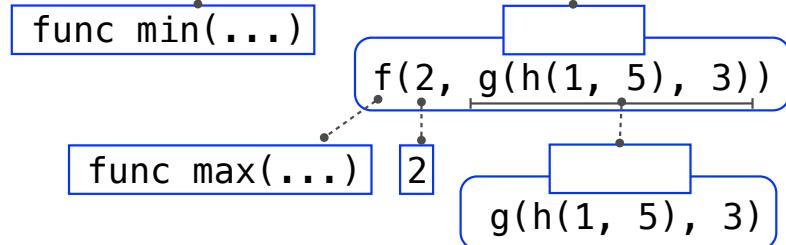
(Demo)



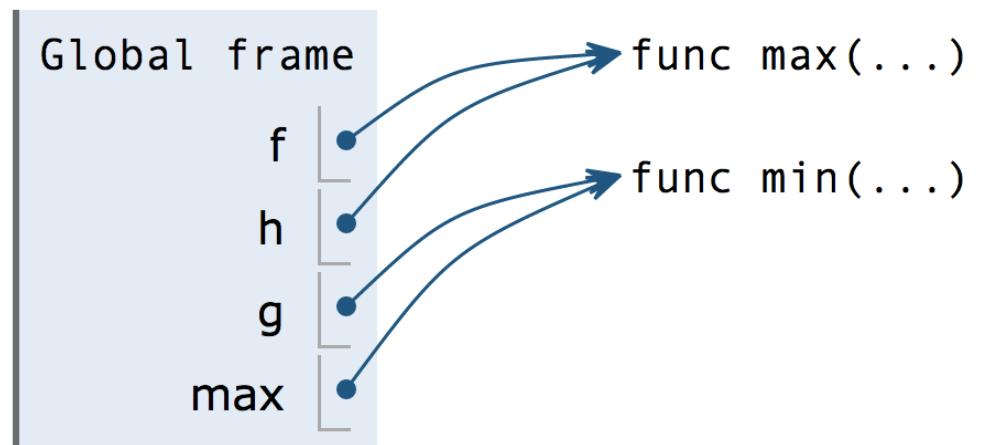
[Interactive Diagram](#)

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)
```



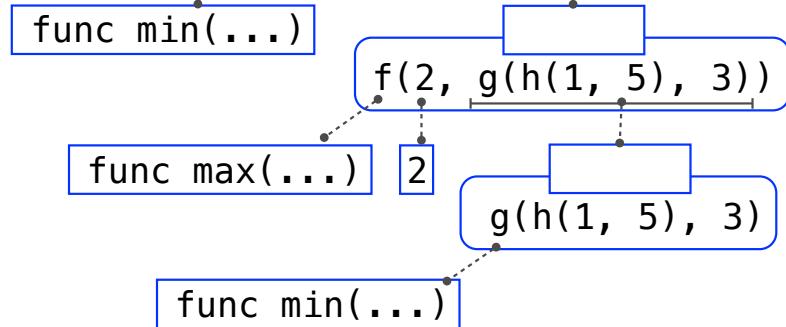
(Demo)



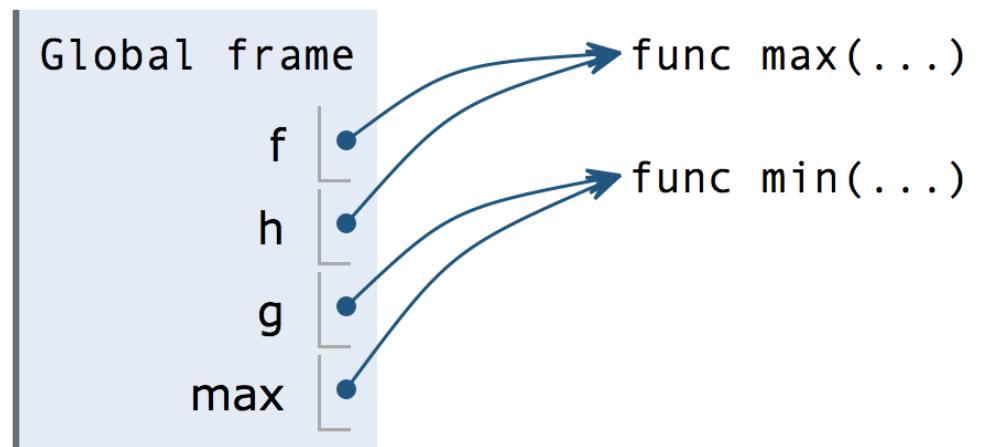
[Interactive Diagram](#)

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)
```

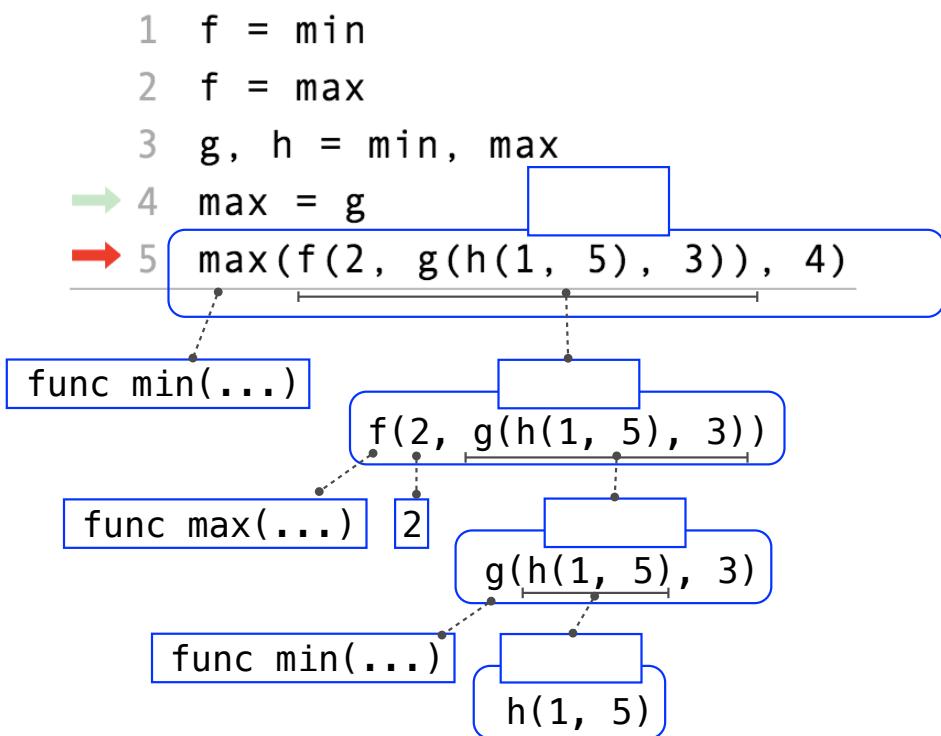


(Demo)

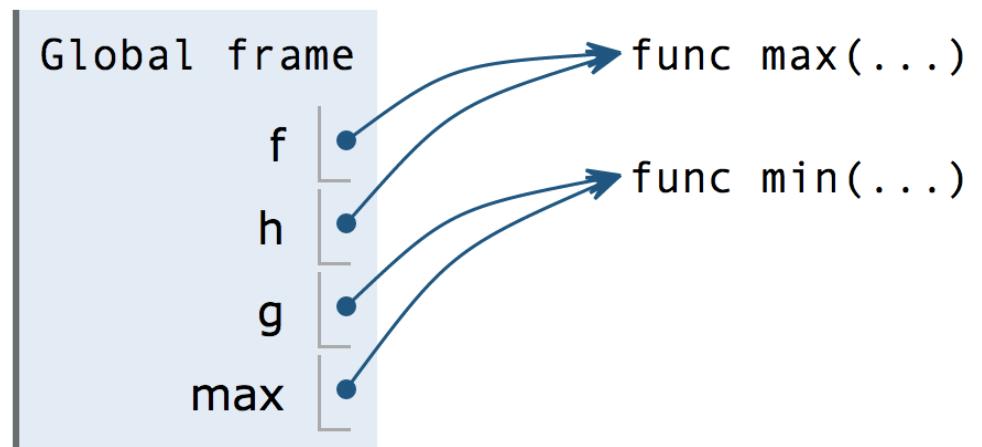


[Interactive Diagram](#)

Discussion Question 1 Solution



(Demo)

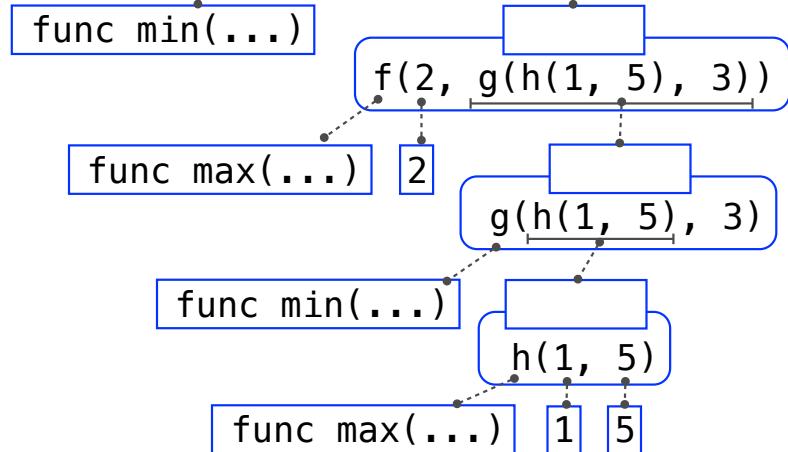


[Interactive Diagram](#)

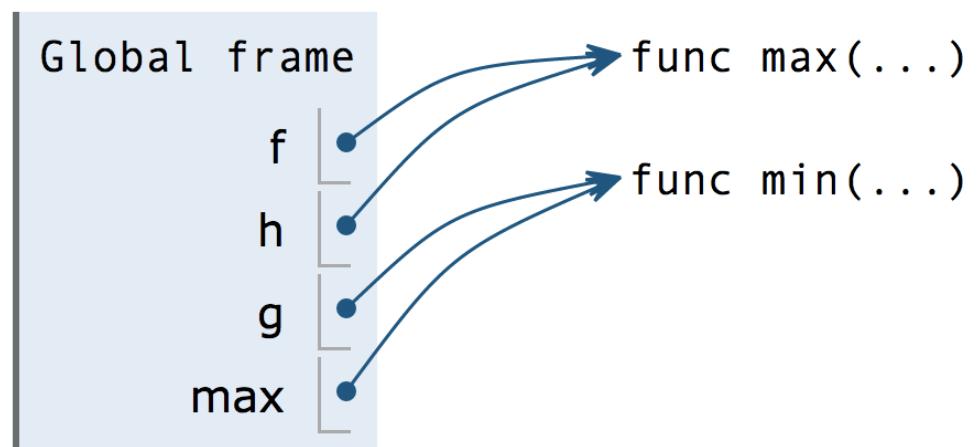
Discussion Question 1 Solution

```

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



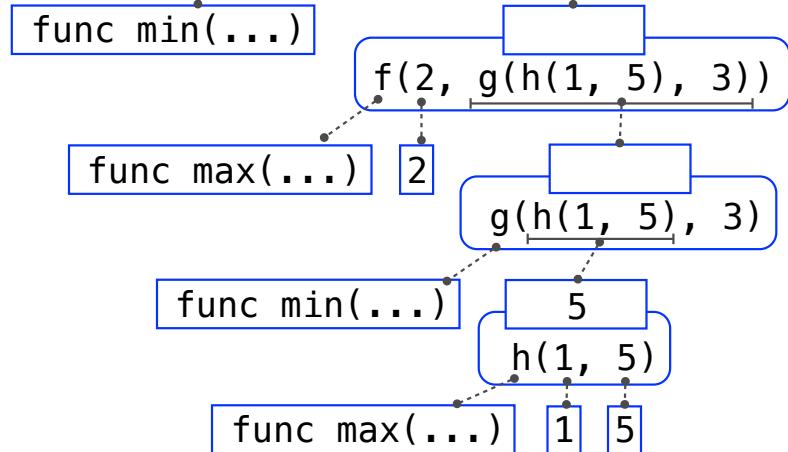
(Demo)



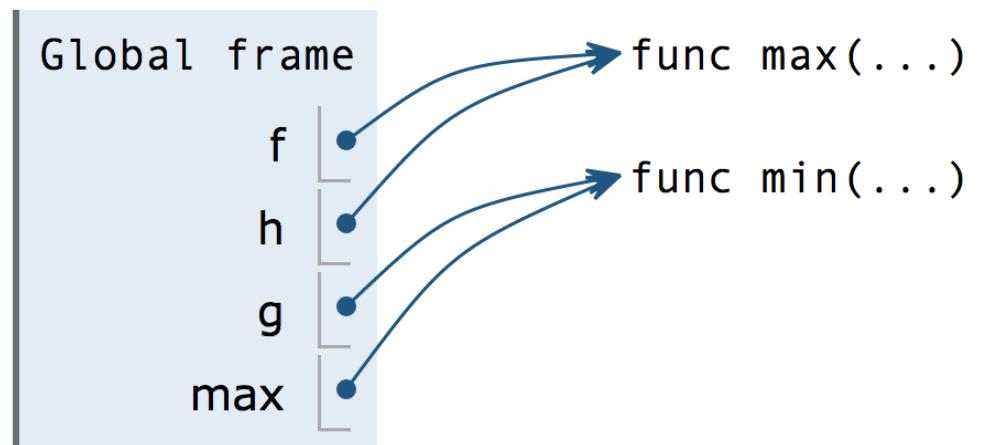
[Interactive Diagram](#)

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)
```



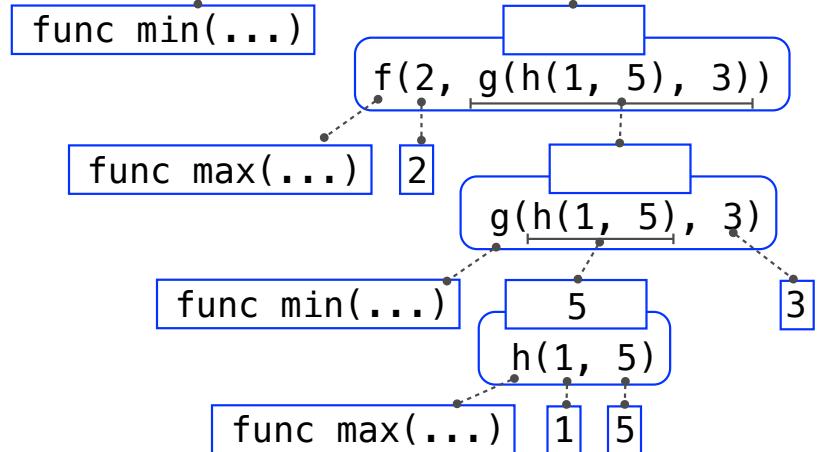
(Demo)



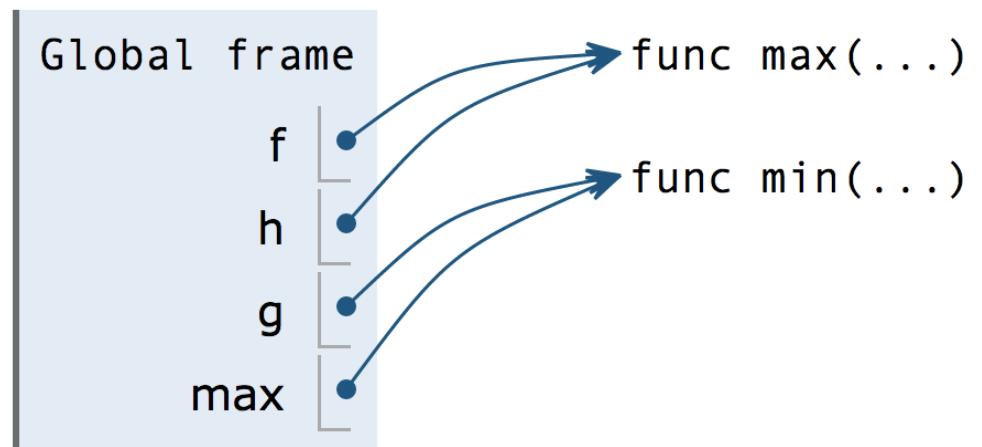
[Interactive Diagram](#)

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)
```



(Demo)

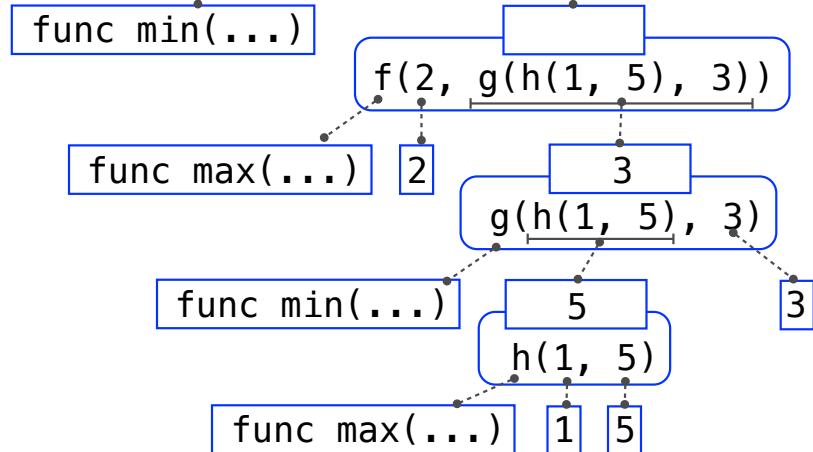


[Interactive Diagram](#)

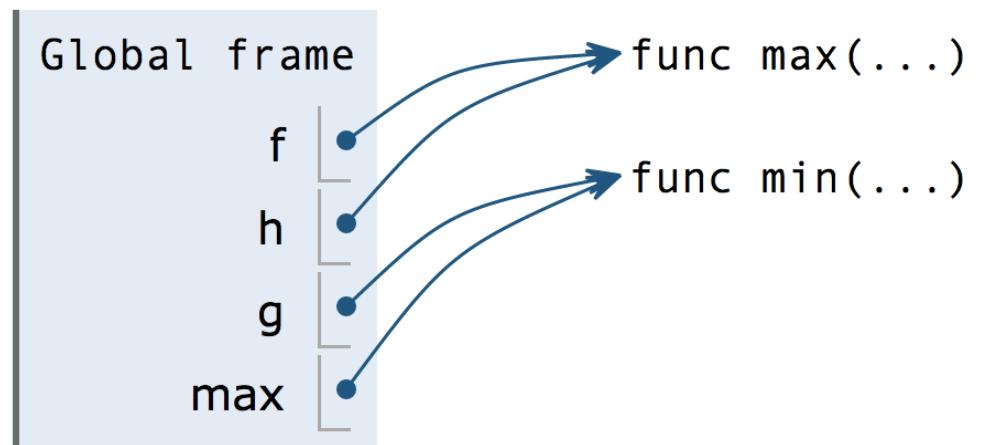
Discussion Question 1 Solution

```

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



(Demo)

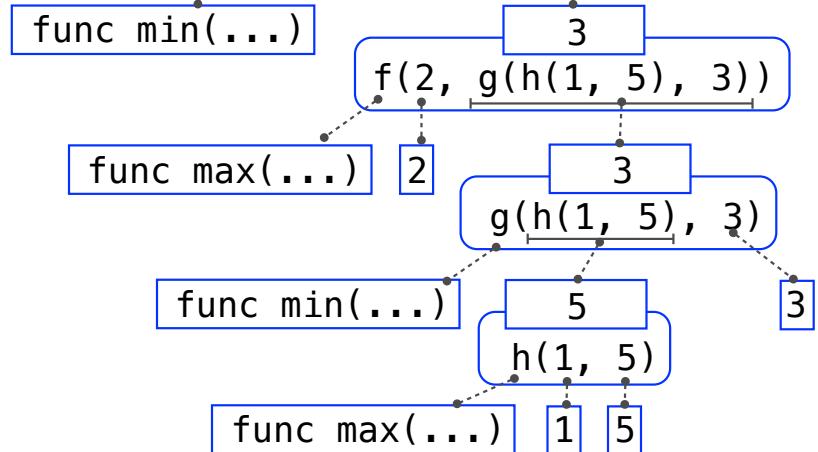


[Interactive Diagram](#)

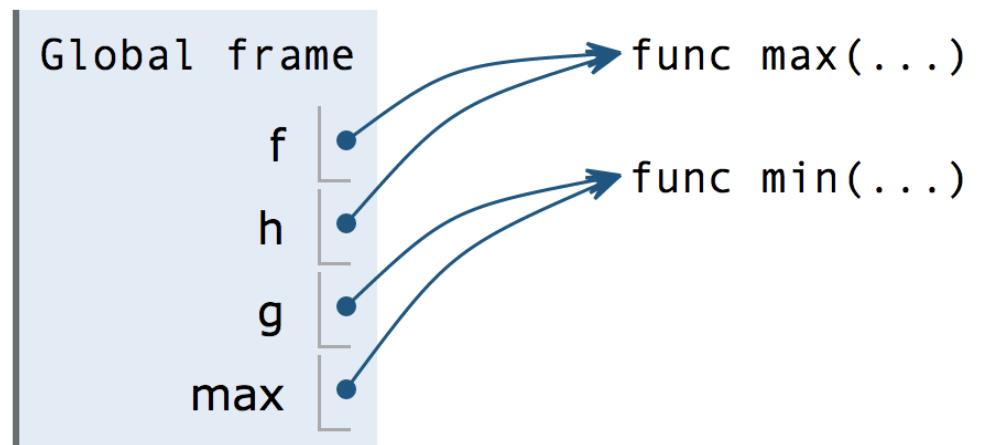
Discussion Question 1 Solution

```

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

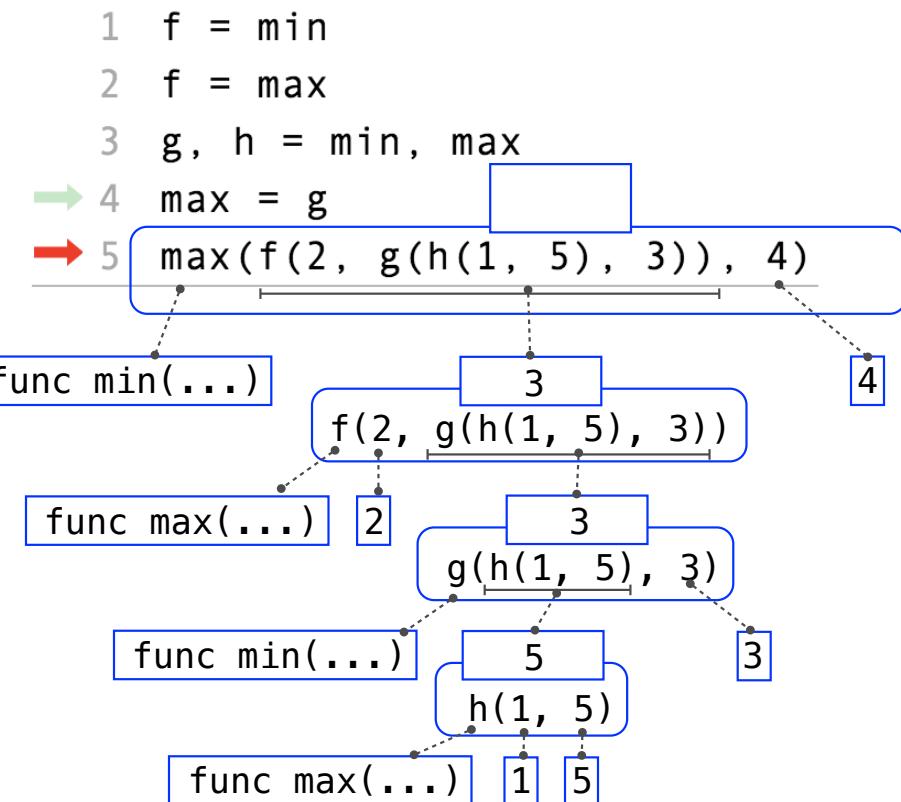


(Demo)

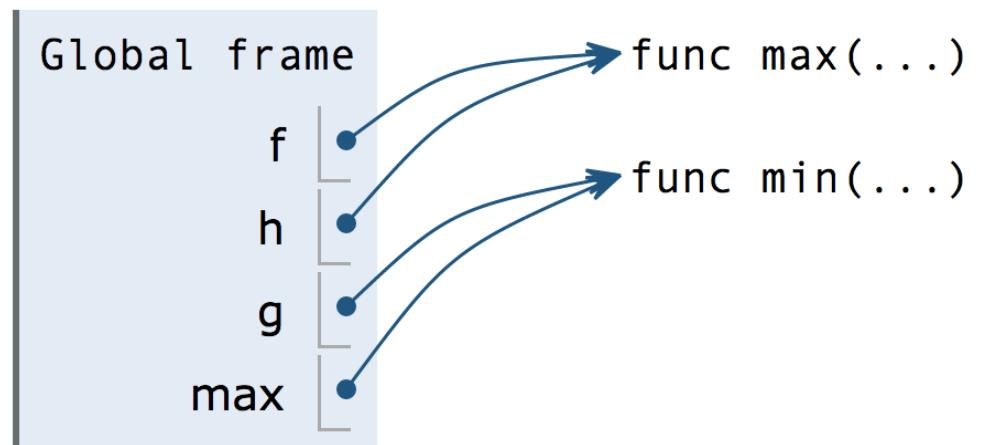


[Interactive Diagram](#)

Discussion Question 1 Solution

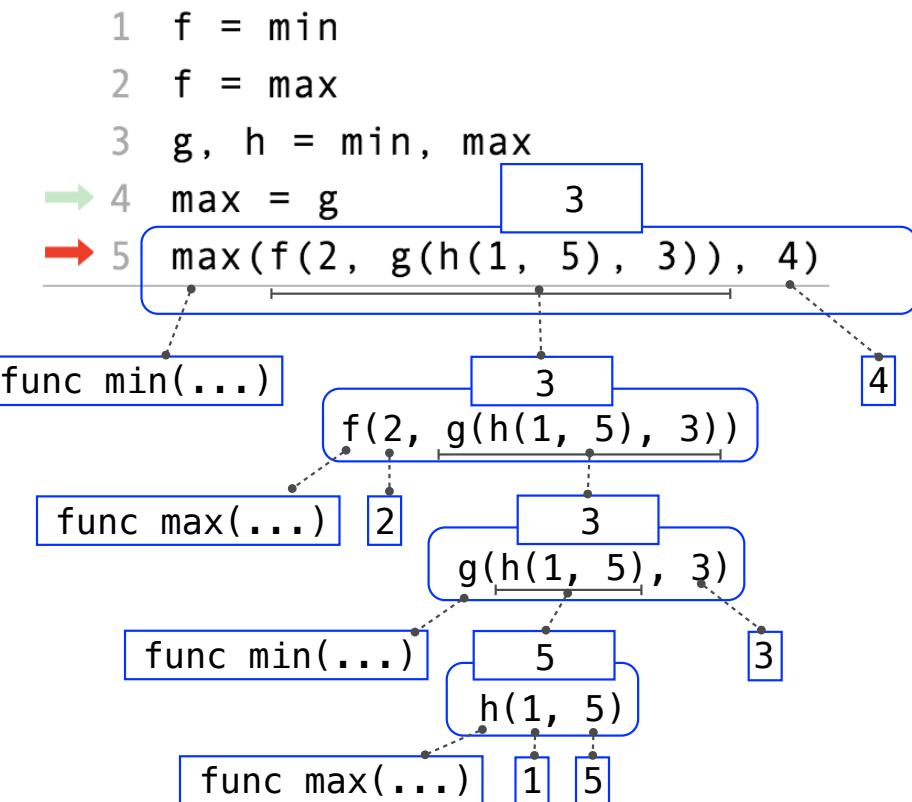


(Demo)

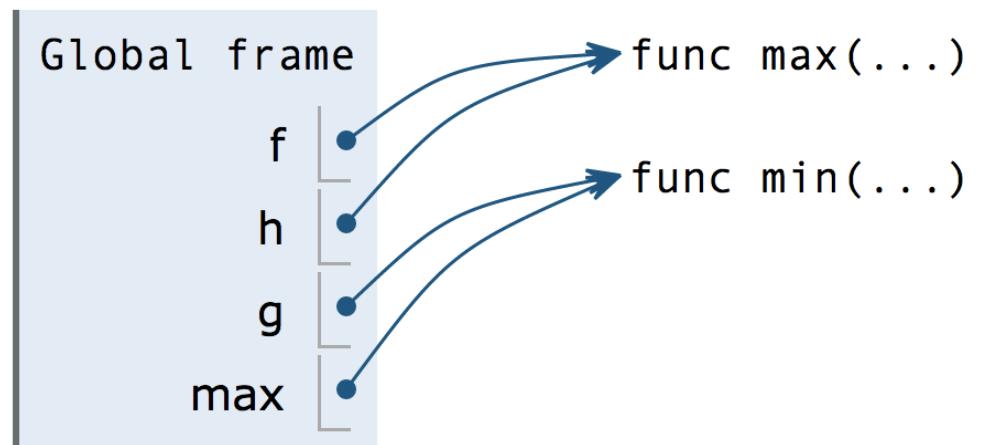


Interactive Diagram

Discussion Question 1 Solution

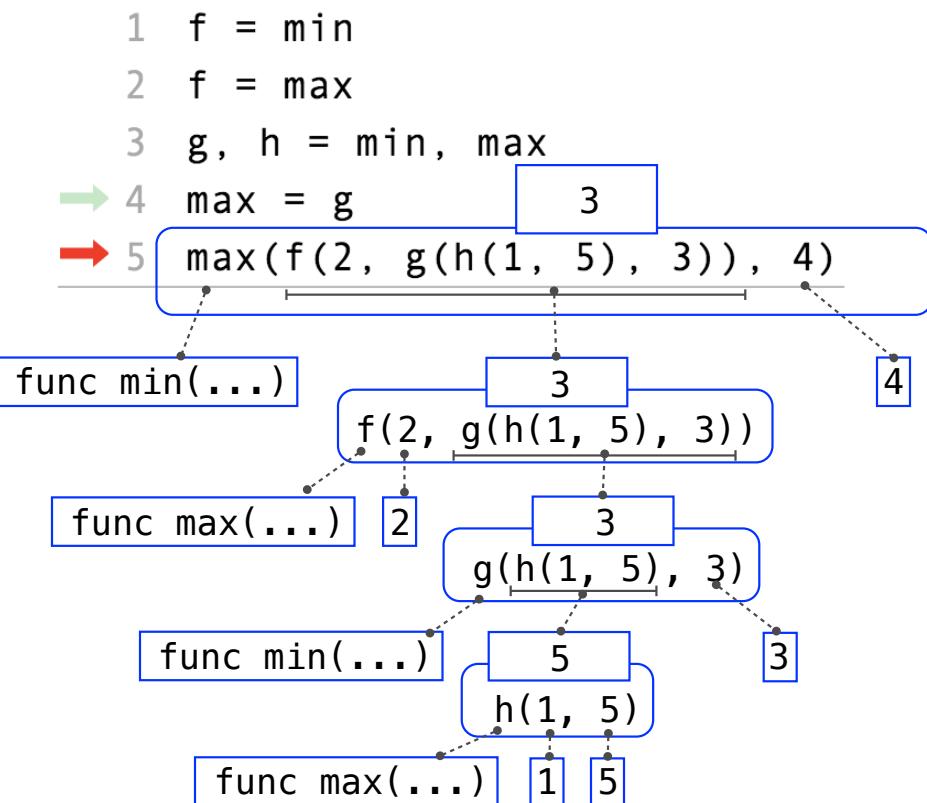


(Demo)

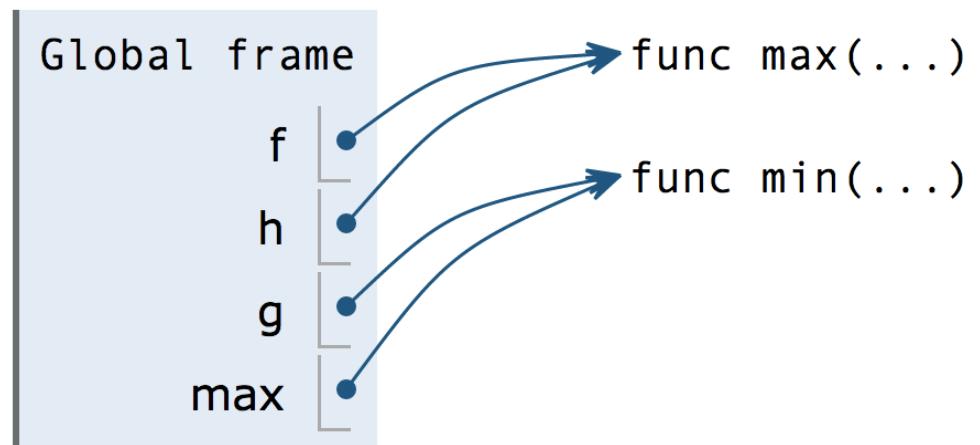


Interactive Diagram

Discussion Question 1 Solution



(Demo)



3

[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

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

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

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>
```

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 expression>
```

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

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 expression>
```

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

Execution procedure for def statements:

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 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>)`

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

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

Calling User-Defined Functions

[Interactive Diagram](#)

12

Calling User-Defined Functions

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

[Interactive Diagram](#)

12

Calling User-Defined Functions

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

1. Add a local frame, forming a new environment

[Interactive Diagram](#)

12

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

[Interactive Diagram](#)

12

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

[Interactive Diagram](#)

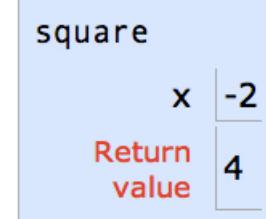
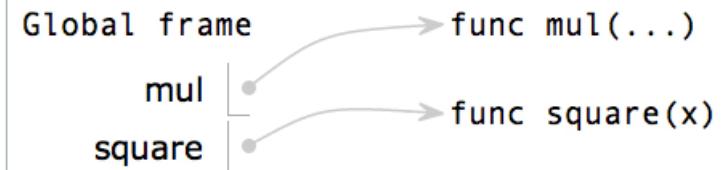
12

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](#)

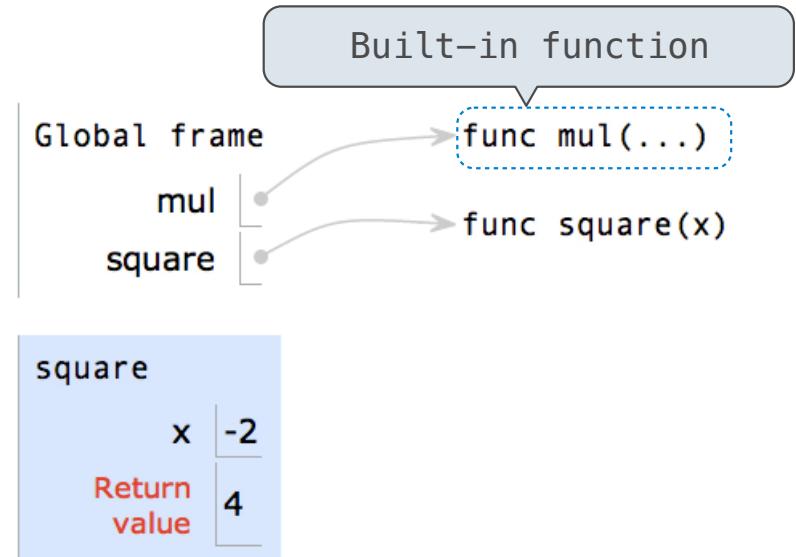
12

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](#)

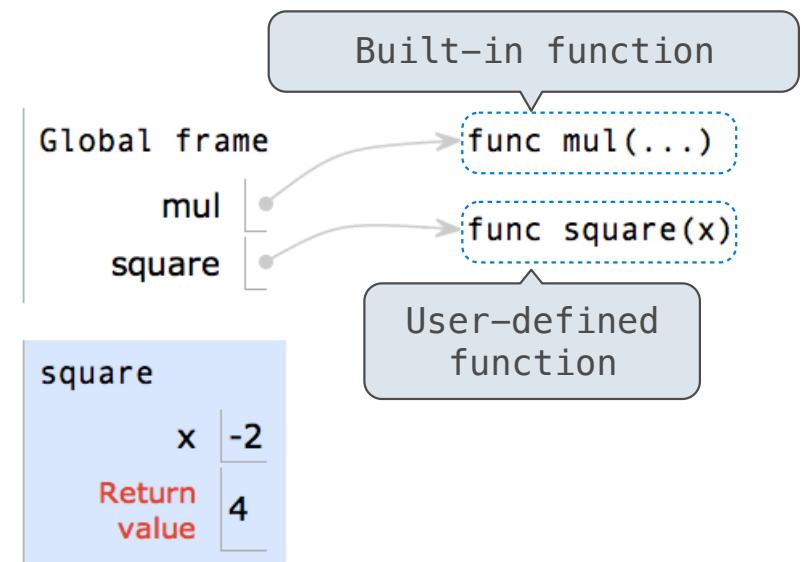
12

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](#)

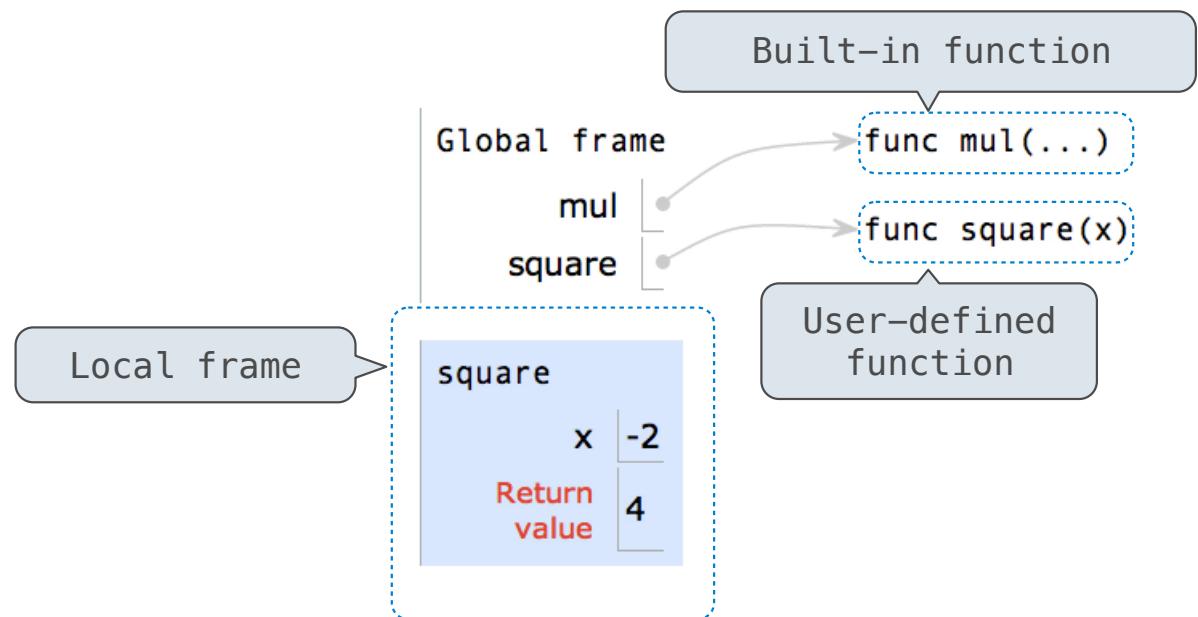
12

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](#)

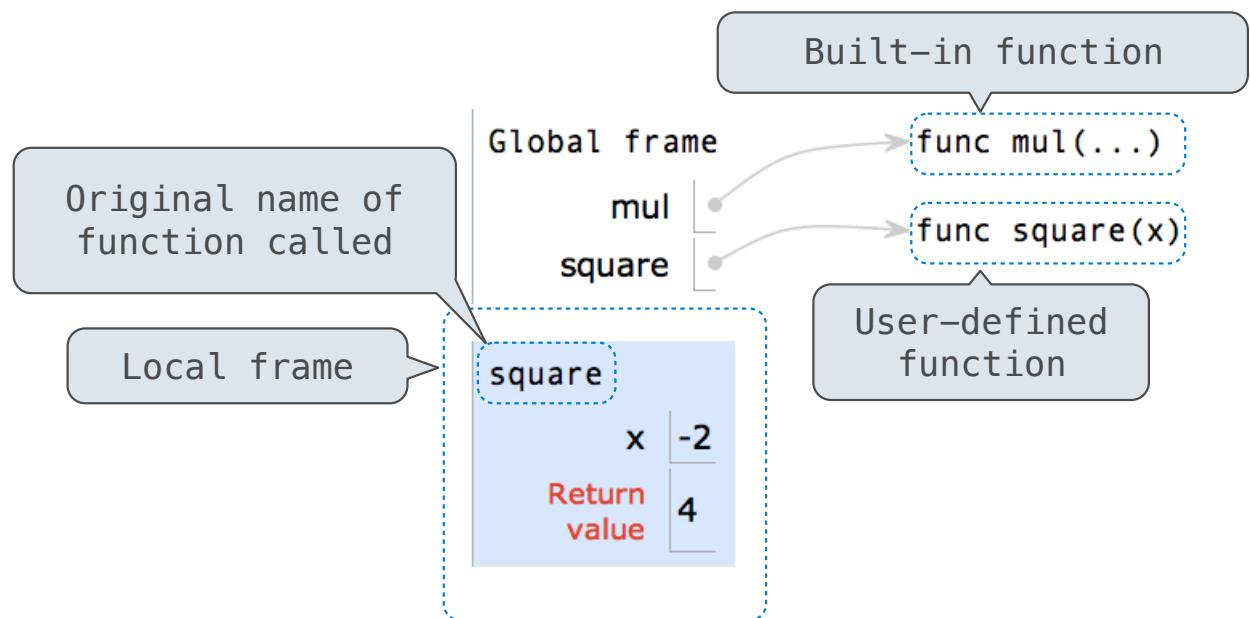
12

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](#)

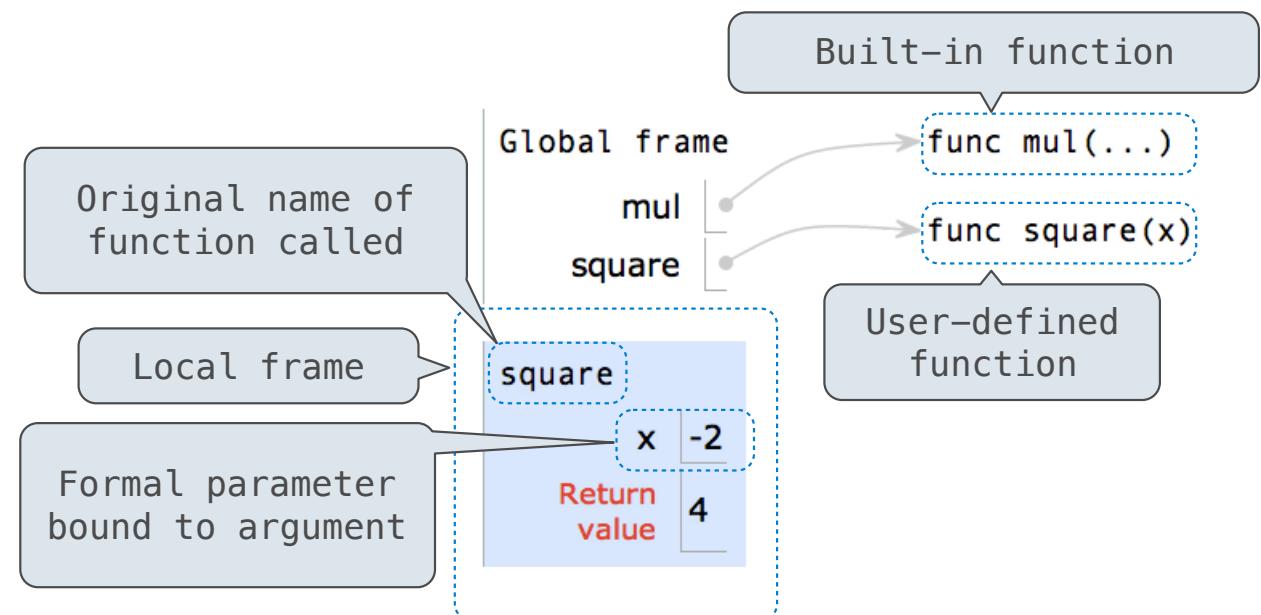
12

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](#)

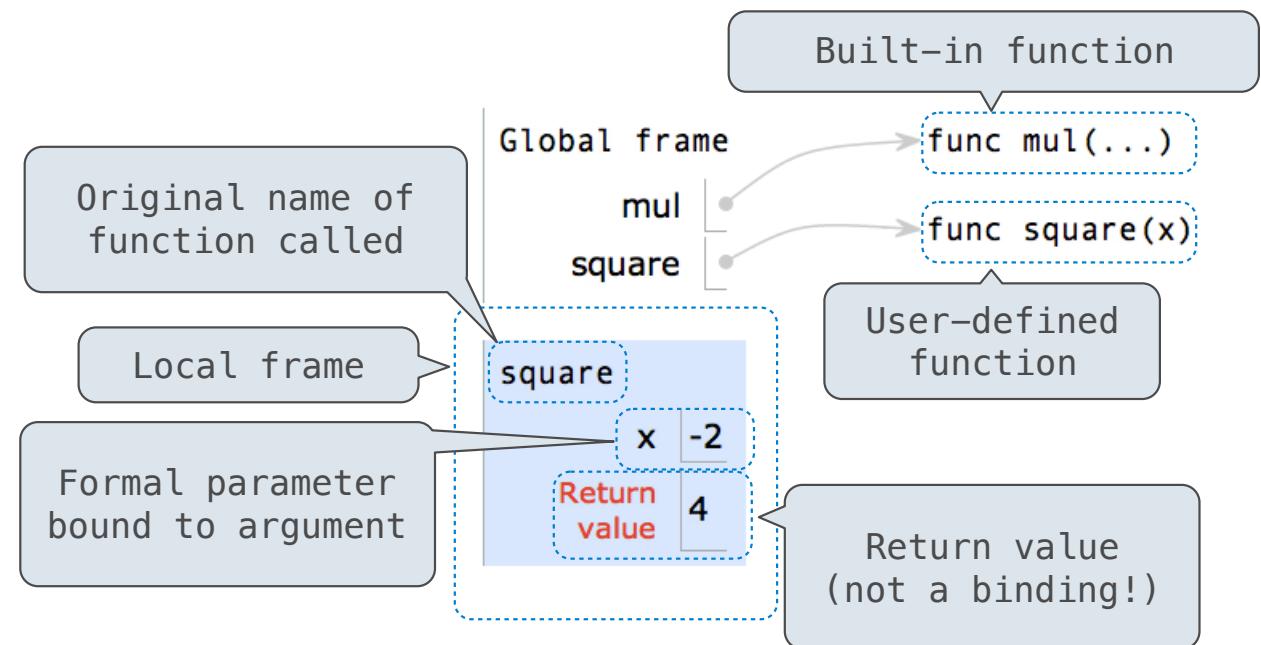
12

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](#)

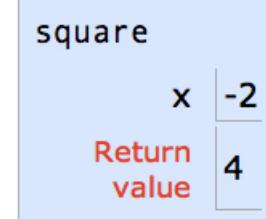
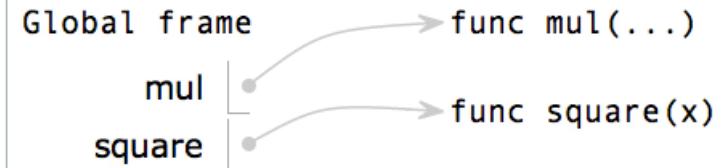
12

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](#)

13

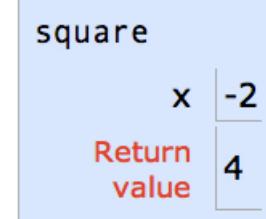
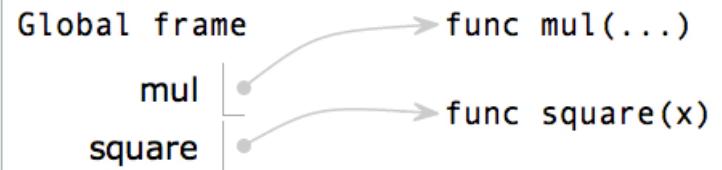
Calling User-Defined Functions

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)
```

A function's signature has all the information needed to create a local frame



[Interactive Diagram](#)

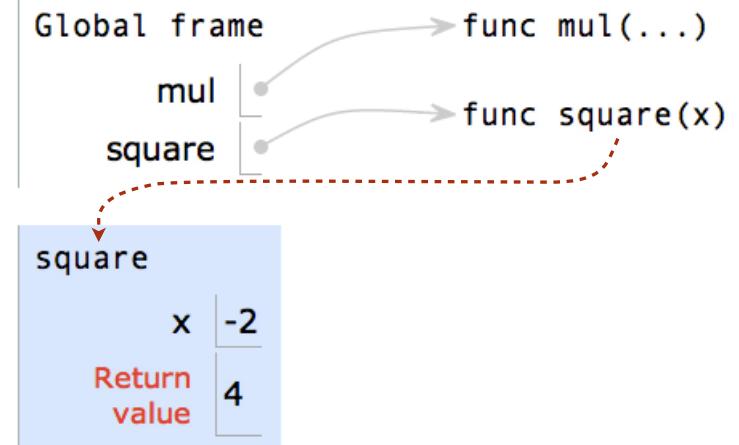
Calling User-Defined Functions

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)
```

A function's signature has all the information needed to create a local frame



[Interactive Diagram](#)

13

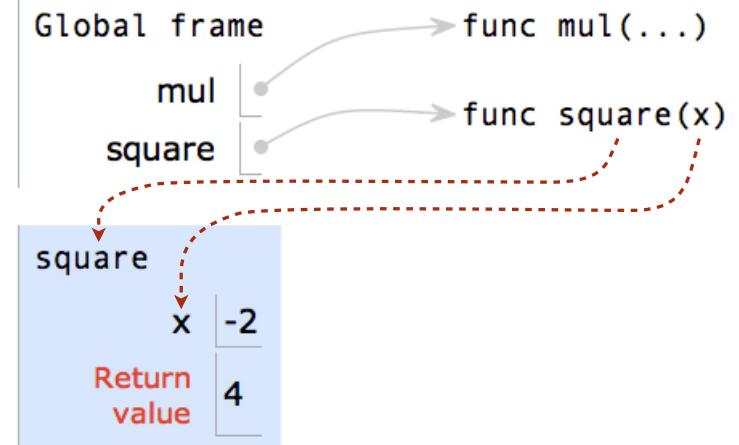
Calling User-Defined Functions

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)
```

A function's signature has all the information needed to create a local frame



[Interactive Diagram](#)

13

Looking Up Names In Environments

Looking Up Names In Environments

Every expression is evaluated in the context of an environment.

Looking Up Names In Environments

Every expression is evaluated in the context of an environment.

So far, the current environment is either:

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

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.

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:

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.

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.

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:

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.

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.)

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

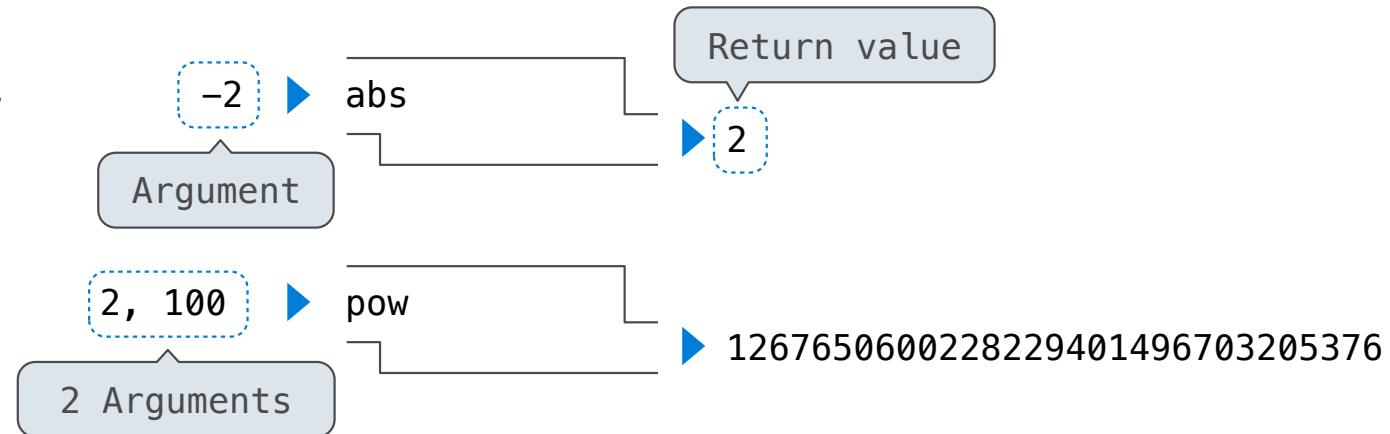
The special value `None` represents nothing in Python

A function that does not explicitly return a value will return **None**

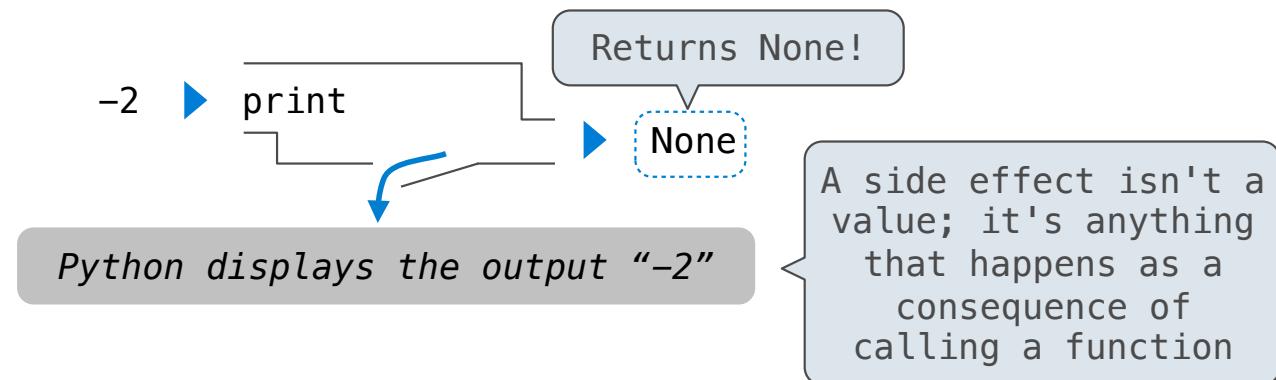
Careful: `None` is not displayed by the interpreter as the value of an expression

Pure Functions & Non-Pure Functions

Pure Functions
just return values



Non-Pure Functions
have side effects



(Demo)

Nested Expressions with Print

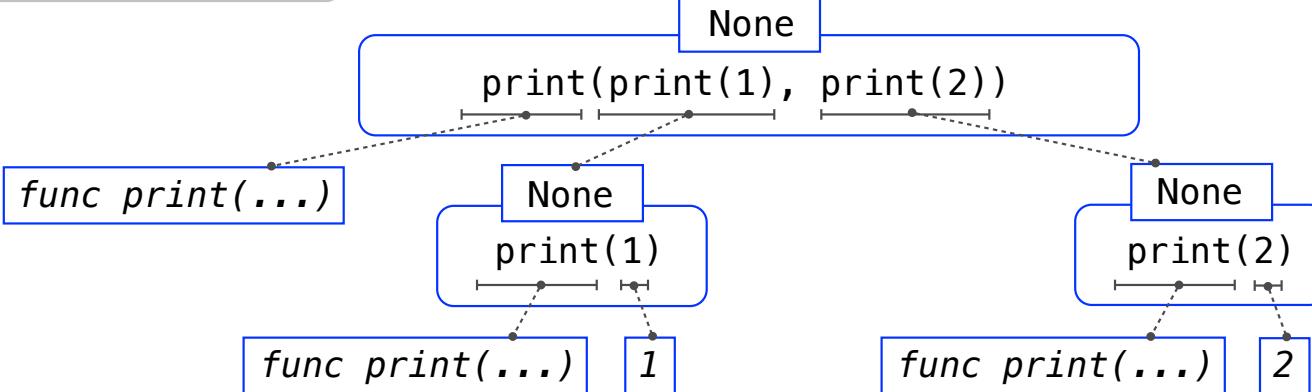
None, None ➤ `print(...):`

None

Does not get displayed

display "None None"

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



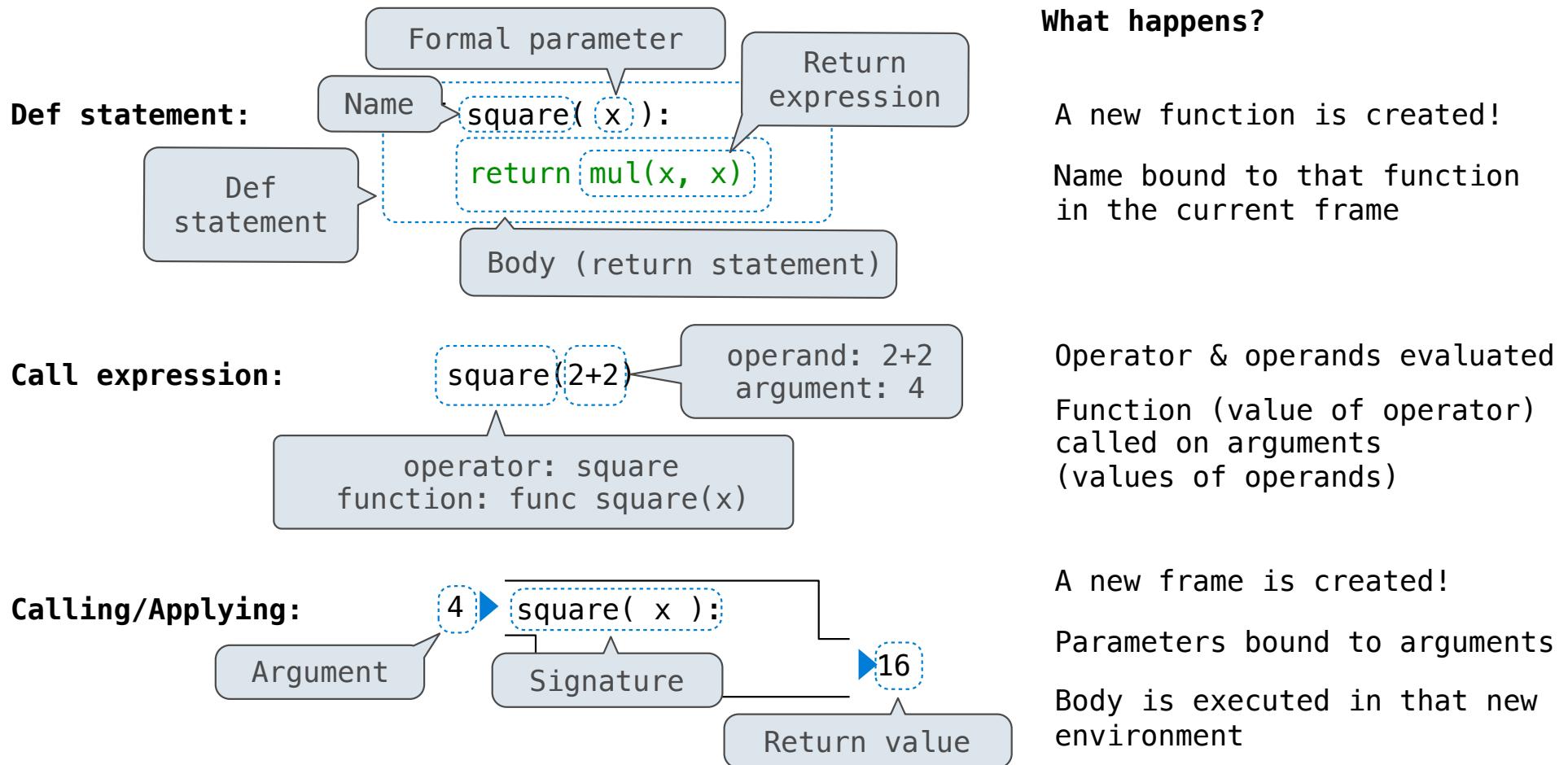
1 ➤ `print(...):` ➤ None

display "1"

2 ➤ `print(...):` ➤ None

display "2"

Life Cycle of a User-Defined Function



Miscellaneous Python Features

Division

Multiple Return Values

Source Files

Doctests

Default Arguments

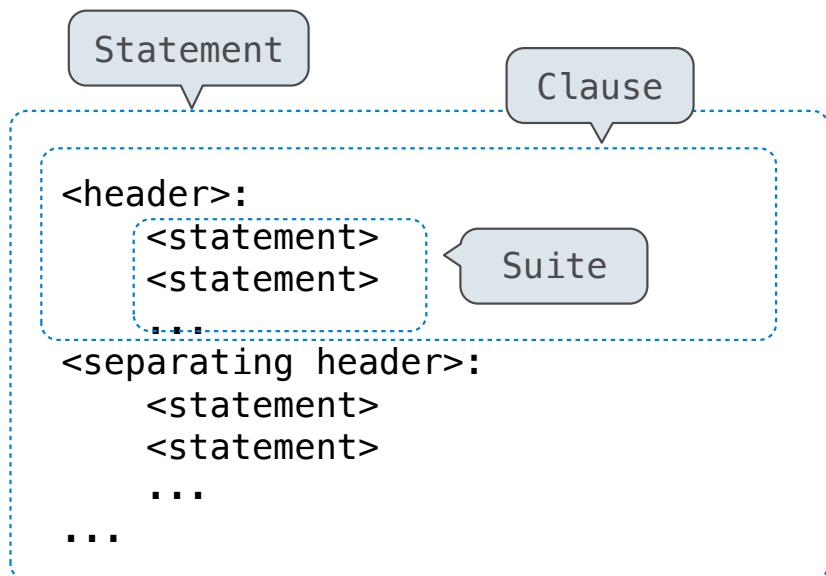
(Demo)

Conditional Statements

Statements

A **statement** is executed by the interpreter to perform an action

Compound statements:



The first header determines a statement's type

The header of a clause “controls” the suite that follows

def statements are compound statements

Compound Statements

Compound statements:

<header>:

<statement>
<statement>
...

Suite

<separating header>:

<statement>
<statement>
...

...

A suite is a sequence of statements

To “execute” a suite means to execute its sequence of statements, in order

Execution Rule for a sequence of statements:

- Execute the first statement
- Unless directed otherwise, execute the rest

Conditional Statements

(Demo)

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

1 statement,
3 clauses,
3 headers,
3 suites

Execution Rule for Conditional Statements:

Each clause is considered in order.

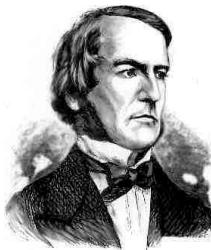
1. Evaluate the header's expression.
2. If it is a true value,
execute the suite & skip the remaining clauses.

Syntax Tips:

1. Always starts with "if" clause.
2. Zero or more "elif" clauses.
3. Zero or one "else" clause,
always at the end.

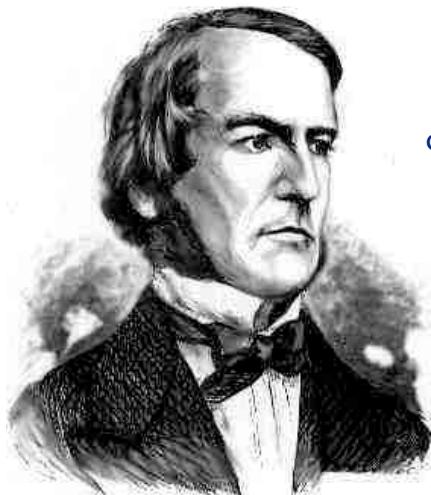
Boolean Contexts

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



George Boole

Boolean Contexts



George Boole

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

Two boolean contexts

False values in Python: False, 0, '', None (*more to come*)

True values in Python: Anything else (True)

Read Section 1.5.4!

While Statements



George Boole

(Demo)

```
▶ 1 i, total = 0, 0
▶ 2 while i < 3:
▶ 3     i = i + 1
▶ 4     total = total + i
```

Global frame
i ✗ ✗ ✗ 3
total ✗ ✗ ✗ 6

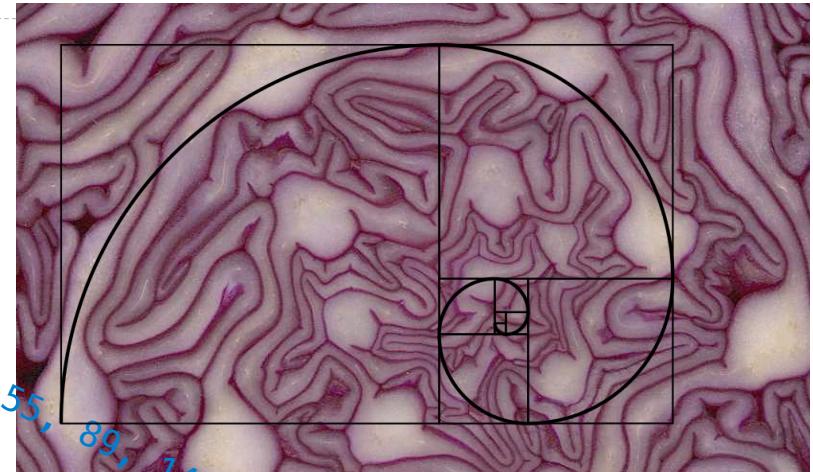
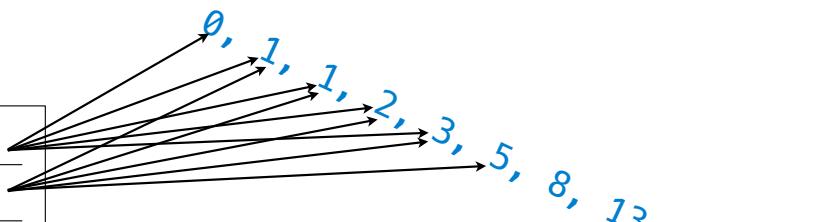
Execution Rule for While Statements:

1. Evaluate the header's expression.
2. If it is a true value,
execute the (whole) suite,
then return to step 1.

Iteration Example

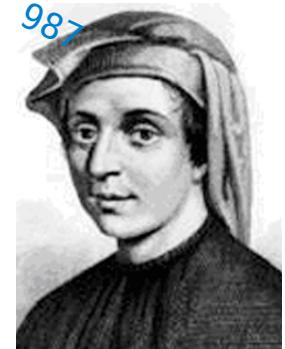
The Fibonacci Sequence

fib	pred	[]
curr		
n	5	
k	5	

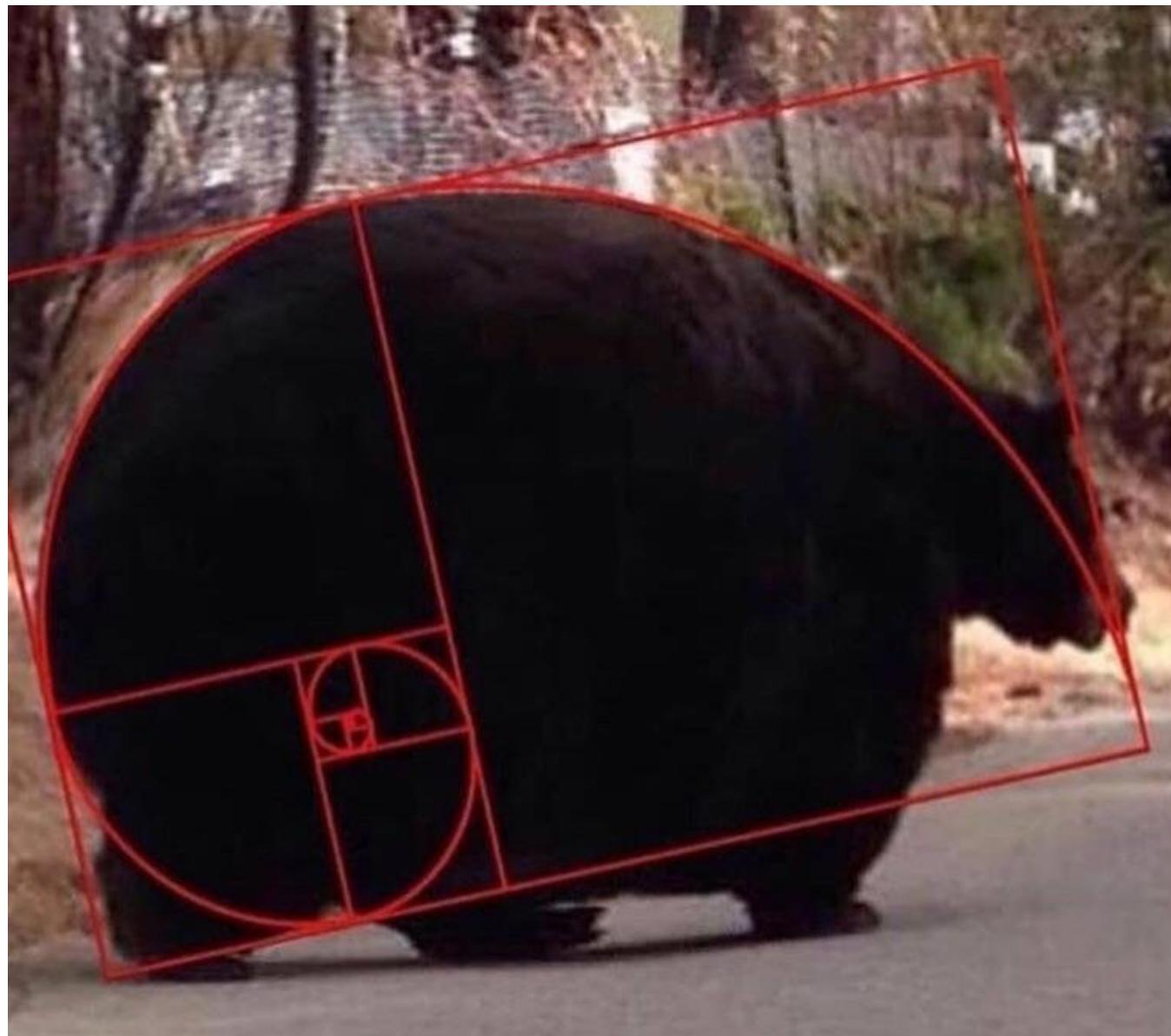


```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1 # 0th and 1st Fibonacci numbers
    k = 1             # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

The next Fibonacci number is the sum of
the current one and its predecessor



Go Bears!



[Return](#)

Return Statements

A return statement completes the evaluation of a call expression and provides its value:

f(x) for user-defined function f: switch to a new environment; execute f's body

`return` statement within f: switch back to the previous environment; f(x) now has a value

Only one return statement is ever executed while executing the body of a function

```
def end(n, d):
    """Print the final digits of N in reverse order until D is found.

    >>> end(34567, 5)
    7
    6
    5
    """
    while n > 0:
        last, n = n % 10, n // 10
        print(last)
        if d == last:
            return None
```

(Demo)

Designing Functions

Describing Functions

A function's *domain* is the set of all inputs it might possibly take as arguments.

A function's *range* is the set of output values it might possibly return.

A pure function's *behavior* is the relationship it creates between input and output.

```
def square(x):  
    """Return X * X."""
```

x is a number

square returns a non-negative real number

square returns the square of x

A Guide to Designing Function

Give each function exactly one job, but make it apply to many related situations

```
>>> round(1.23)      >>> round(1.23, 1)      >>> round(1.23, 0)      >>> round(1.23, 5)  
1                      1.2                      1                      1.23
```

Don't repeat yourself (DRY): Implement a process just once, but execute it many times

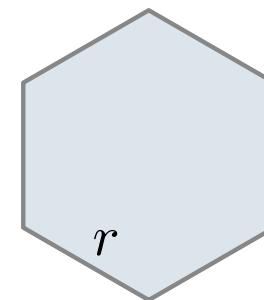
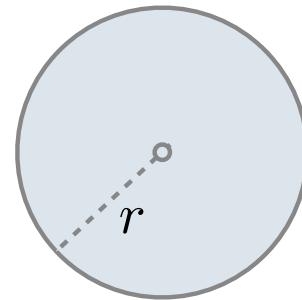
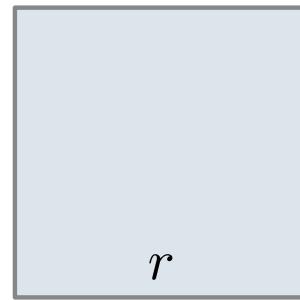
(Demo)

Generalization

Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Shape:



Area:

$$\boxed{1} \cdot r^2$$

$$\boxed{\pi} \cdot r^2$$

$$\boxed{\frac{3\sqrt{3}}{2}} \cdot r^2$$

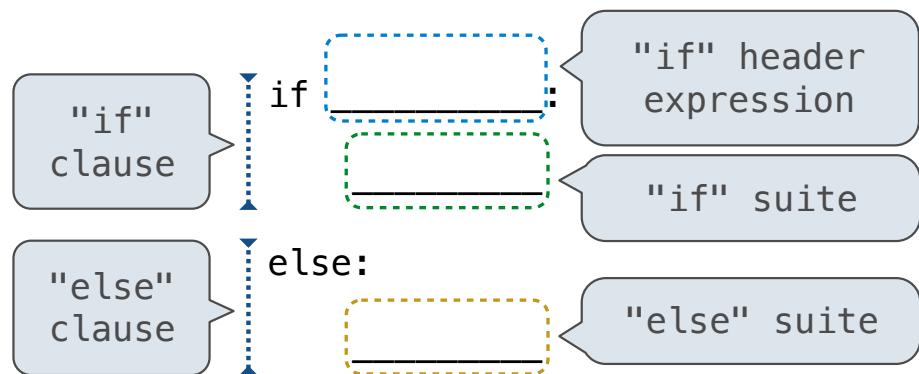
Finding common structure allows for shared implementation

(Demo)

Control

If Statements and Call Expressions

Let's try to write a function that does the same thing as an if statement.



Execution Rule for Conditional Statements:

Each clause is considered in order.

1. Evaluate the header's expression (if present).
2. If it is a true value (or an else header), execute the suite & skip the remaining clauses.

(Demo)

This function doesn't exist



"if" header expression

"if" suite

"else" suite

Evaluation Rule 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

Control Expressions

Logical Operators

To evaluate the expression **<left> and <right>**:

1. Evaluate the subexpression **<left>**.
2. If the result is a false value **v**, then the expression evaluates to **v**.
3. Otherwise, the expression evaluates to the value of the subexpression **<right>**.

To evaluate the expression **<left> or <right>**:

1. Evaluate the subexpression **<left>**.
2. If the result is a true value **v**, then the expression evaluates to **v**.
3. Otherwise, the expression evaluates to the value of the subexpression **<right>**.

(Demo)

Conditional Expressions

A conditional expression has the form

```
<consequent> if <predicate> else <alternative>
```

Evaluation rule:

1. Evaluate the **<predicate>** expression.
2. If it's a true value, the value of the whole expression is the value of the **<consequent>**.
3. Otherwise, the value of the whole expression is the value of the **<alternative>**.

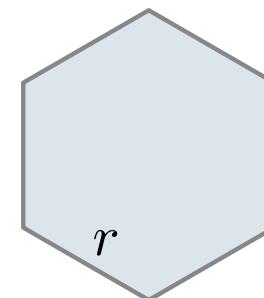
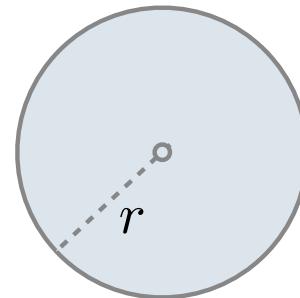
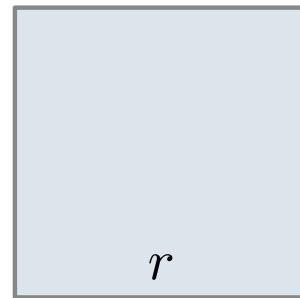
```
>>> x = 0
>>> abs(1/x if x != 0 else 0)
0
```

Higher-Order Functions

Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Shape:



Area:

$$\boxed{1} \cdot r^2$$

$$\boxed{\pi} \cdot r^2$$

$$\boxed{\frac{3\sqrt{3}}{2}} \cdot r^2$$

Finding common structure allows for shared implementation

(Demo)

Generalizing Over Computational Processes

The common structure among functions may be a computational process, rather than a number.

$$\sum_{k=1}^5 k = 1 + 2 + 3 + 4 + 5 = 15$$

$$\sum_{k=1}^5 k^3 = 1^3 + 2^3 + 3^3 + 4^3 + 5^3 = 225$$

$$\sum_{k=1}^5 \frac{8}{(4k-3) \cdot (4k-1)} = \frac{8}{3} + \frac{8}{35} + \frac{8}{99} + \frac{8}{195} + \frac{8}{323} = 3.04$$

(Demo)

Summation Example

```
def cube(k):
    return pow(k, 3)
```

Function of a single argument
(not called "term")

```
def summation(n, term)
    """Sum the first n terms of a sequence.
```

A formal parameter that will
be bound to a function

```
>>> summation(5, cube)
225
"""
total, k = 0, 1
while k <= n:
    total, k = total + term(k), k + 1
return total
```

The cube function is passed
as an argument value

$0 + 1 + 8 + 27 + 64 + 125$

The function bound to term
gets called here

Functions as Return Values

(Demo)

Locally Defined Functions

Functions defined within other function bodies are bound to names in a local frame

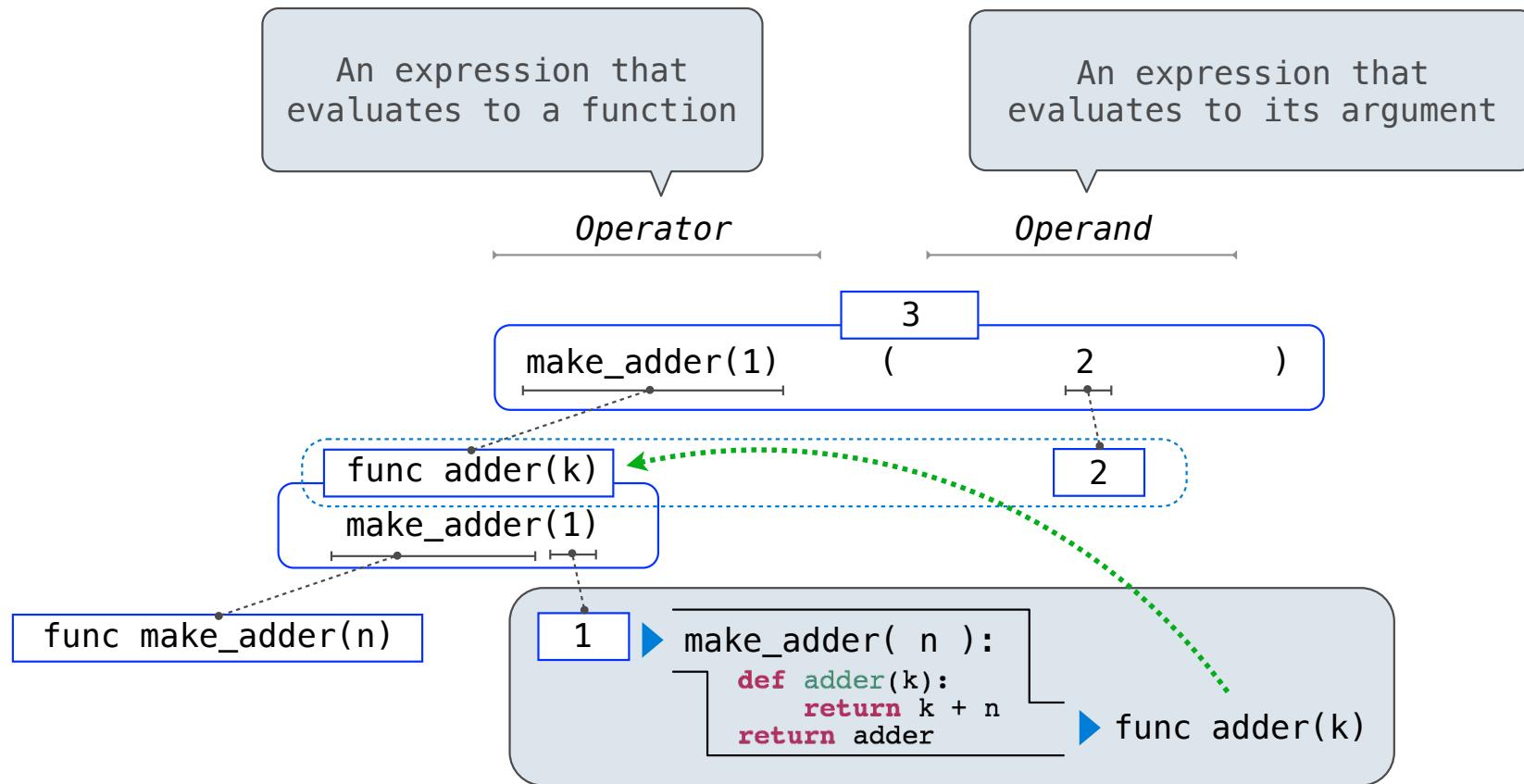
```
A function that  
returns a function  
  
def make_adder(n):  
    """Return a function that takes one argument k and returns k + n.  
  
    >>> add_three = make_adder(3)  
    >>> add_three(4)  
    7  
    """  
def adder(k):  
    return k + n  
return adder
```

The name add_three is bound to a function

A def statement within another def statement

Can refer to names in the enclosing function

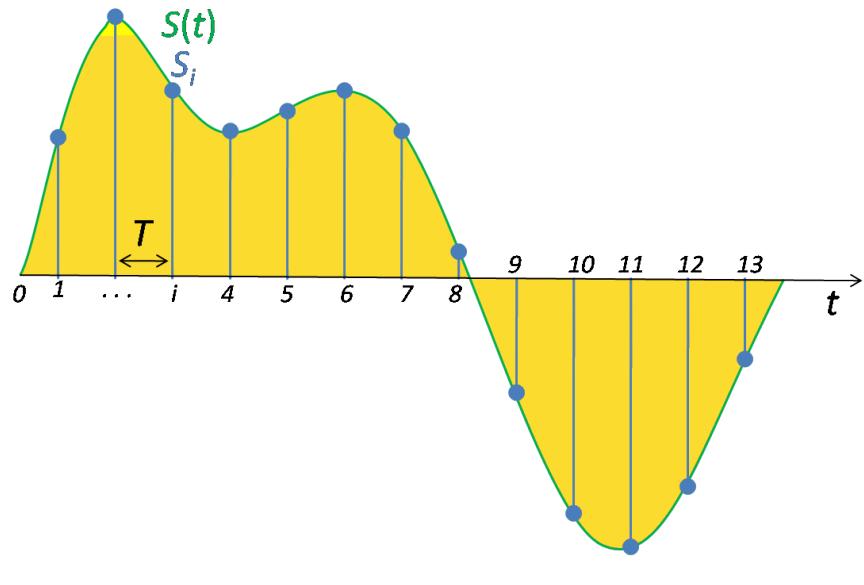
Call Expressions as Operator Expressions



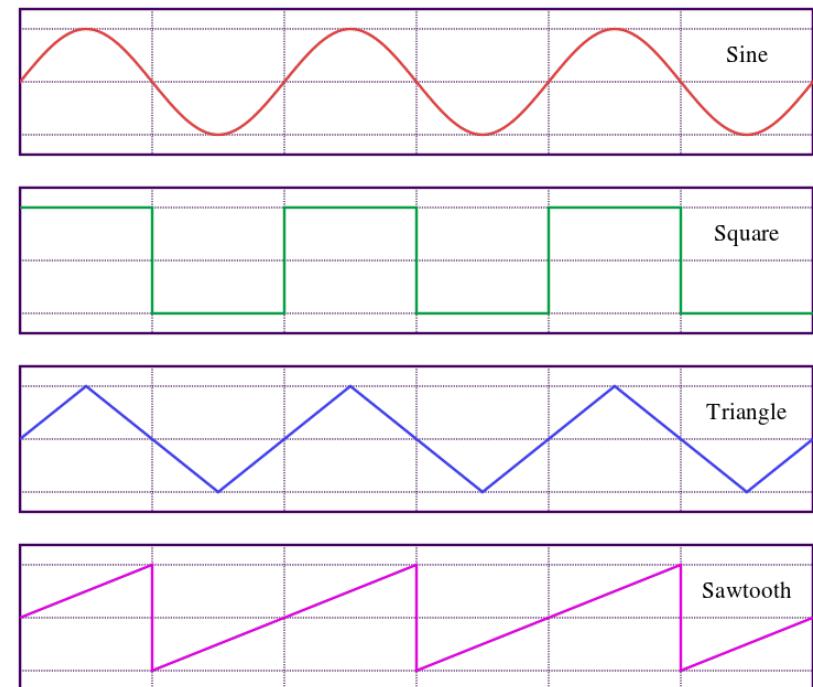
Function Example: Sounds

WAV Files

The Waveform Audio File Format encodes a sampled sound wave



A triangle wave is the simple waveform with the most pleasing sound



(Demo)

https://en.wikipedia.org/wiki/Triangle_wave
[https://en.wikipedia.org/wiki/Sampling_\(signal_processing\)](https://en.wikipedia.org/wiki/Sampling_(signal_processing))

Function Composition

(Demo)

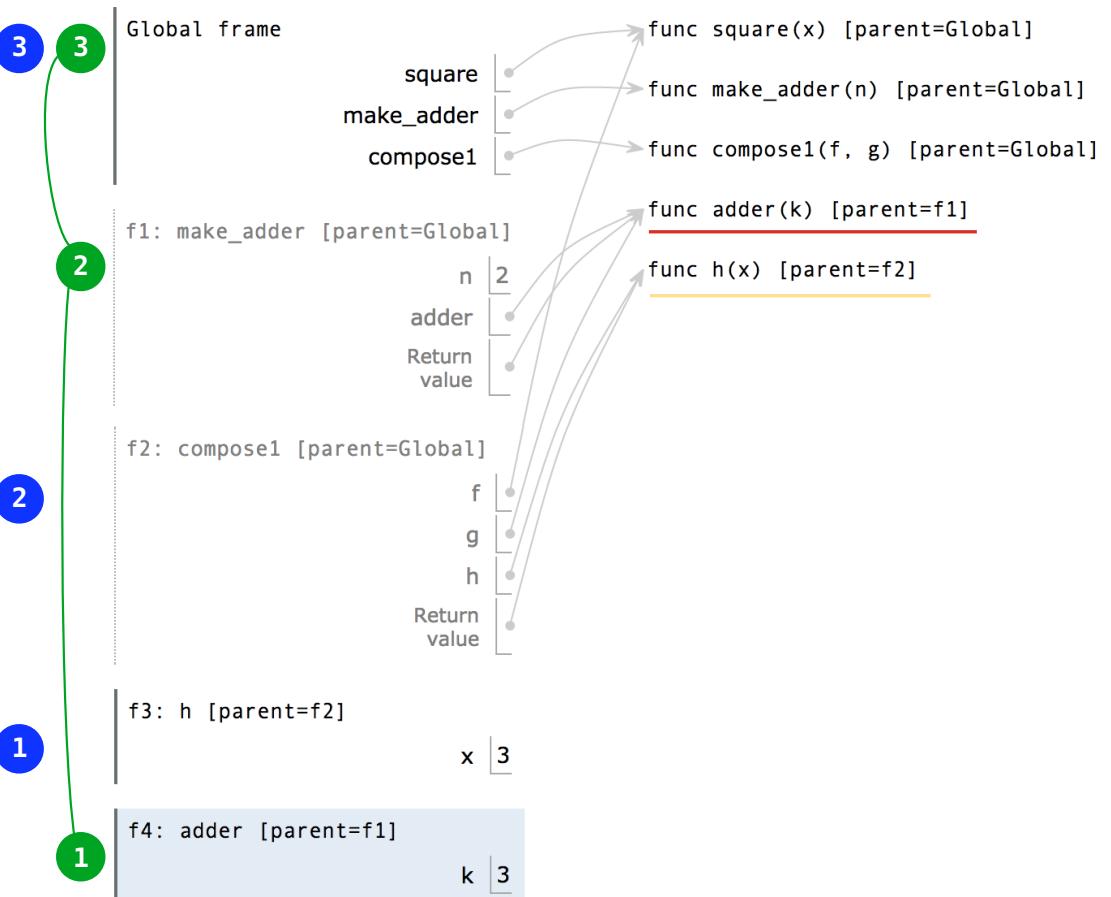
The Environment Diagram for Function Composition

```

1 def square(x):
2     return x * x
3
4 def make_adder(n):
5     def adder(k):
6         return k + n
7     return adder
8
9 def compose1(f, g):
10    def h(x):
11        return f(g(x))
12    return h
13
14 compose1(square, make_adder(2))(3)

```

Return value of `make_adder` is
an argument to `compose1`



http://pythontutor.com/composingprograms.html#code=def%20square%28x%29%3A%0A%20%20%20return%20x%20*x%0Adef%20make_adder%28n%3A%0A%20%20%20def%20adder%28k%3A%0A%20%20%20def%20h%28x%29%3A%0A%20%20%20return%20h%20+k%29&mode=display&origin=composingprograms.js&rawInputStJSON=%5B%5D

Abstraction

Functional Abstractions

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

```
def sum_squares(x, y):  
    return square(x) + square(y)
```

What does `sum_squares` need to know about `square`?

- Square takes one argument. Yes
- Square has the intrinsic name `square`. No
- Square computes the square of a number. Yes
- Square computes the square by calling `mul`. No

```
def square(x):  
    return pow(x, 2)
```

```
def square(x):  
    return mul(x, x-1) + x
```

If the name “`square`” were bound to a built-in function,
`sum_squares` would still work identically.

Choosing Names

Names typically don't matter for correctness

but

they matter a lot for composition

From:

true_false

d

helper

my_int

l, I, 0

To:

rolled_a_one

dice

take_turn

num_rolls

k, i, m

Names should convey the meaning or purpose of the values to which they are bound.

The type of value bound to the name is best documented in a function's docstring.

Function names typically convey their effect (**print**), their behavior (**triple**), or the value returned (**abs**).

Which Values Deserve a Name

Reasons to add a new name

Repeated compound expressions:

```
if sqrt(square(a) + square(b)) > 1:  
    x = x + sqrt(square(a) + square(b))
```



```
hypotenuse = sqrt(square(a) + square(b))  
if hypotenuse > 1:  
    x = x + hypotenuse
```

PRACTICAL
GUIDELINES

Meaningful parts of complex expressions:

```
x1 = (-b + sqrt(square(b) - 4 * a * c)) / (2 * a)
```



```
discriminant = square(b) - 4 * a * c  
x1 = (-b + sqrt(discriminant)) / (2 * a)
```

More Naming Tips

- Names can be long if they help document your code:

```
average_age = average(age, students)
```

is preferable to

```
# Compute average age of students  
aa = avg(a, st)
```

- Names can be short if they represent generic quantities: counts, arbitrary functions, arguments to mathematical operations, etc.

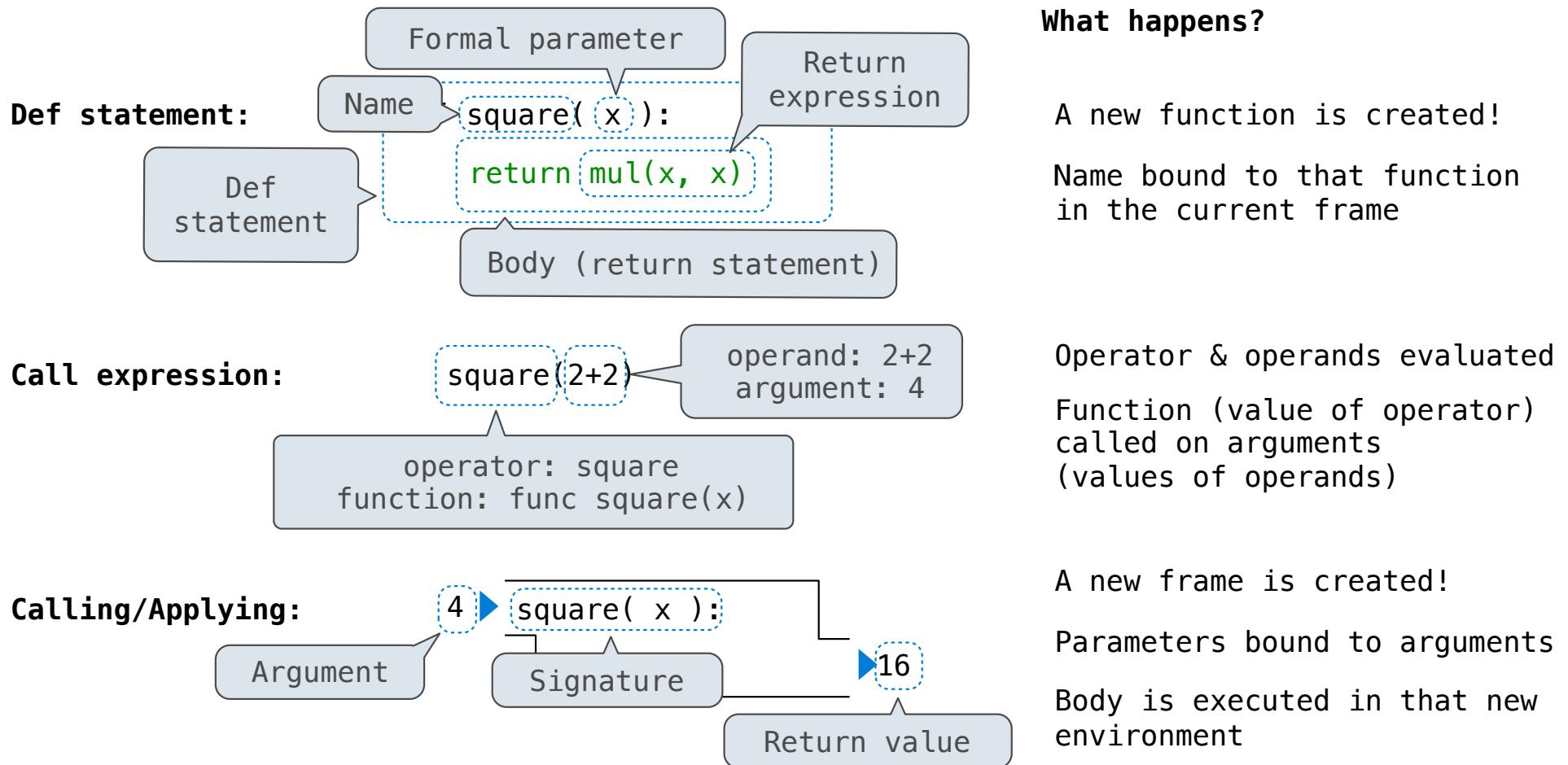
n, k, i – Usually integers

x, y, z – Usually real numbers

f, g, h – Usually functions

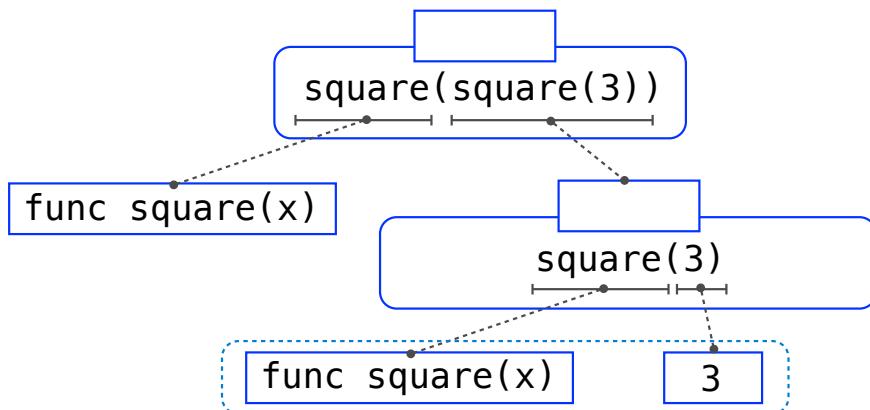
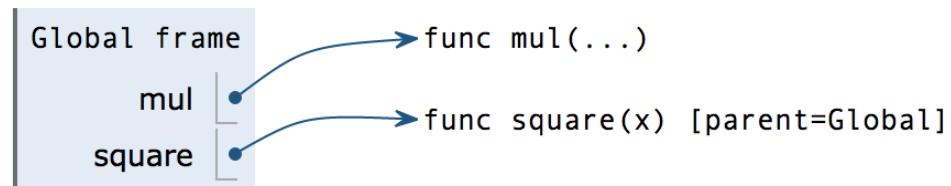
Multiple Environments

Life Cycle of a User-Defined Function



Multiple Environments in One Diagram!

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

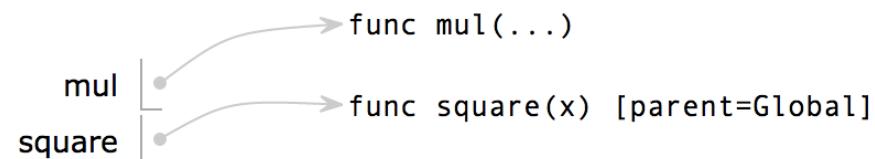


Interactive Diagram

Multiple Environments in One Diagram!

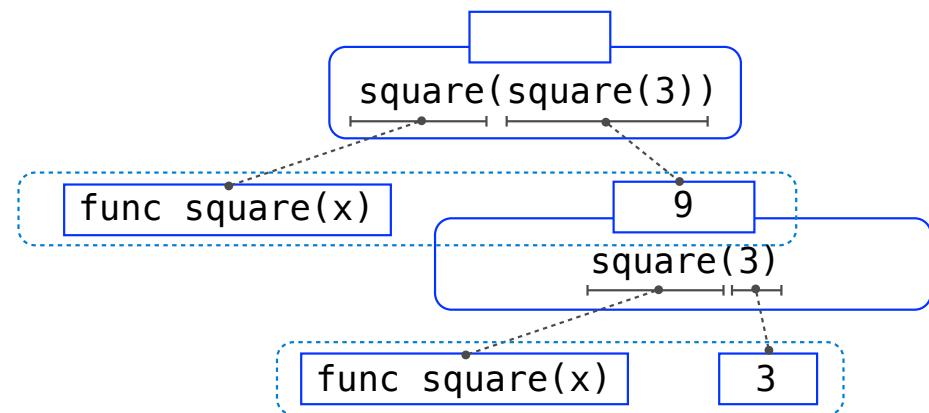
```
1 from operator import mul  
2 def square(x):  
3     return mul(x, x)  
4 square(square(3))
```

Global frame



f1: square [parent=Global]

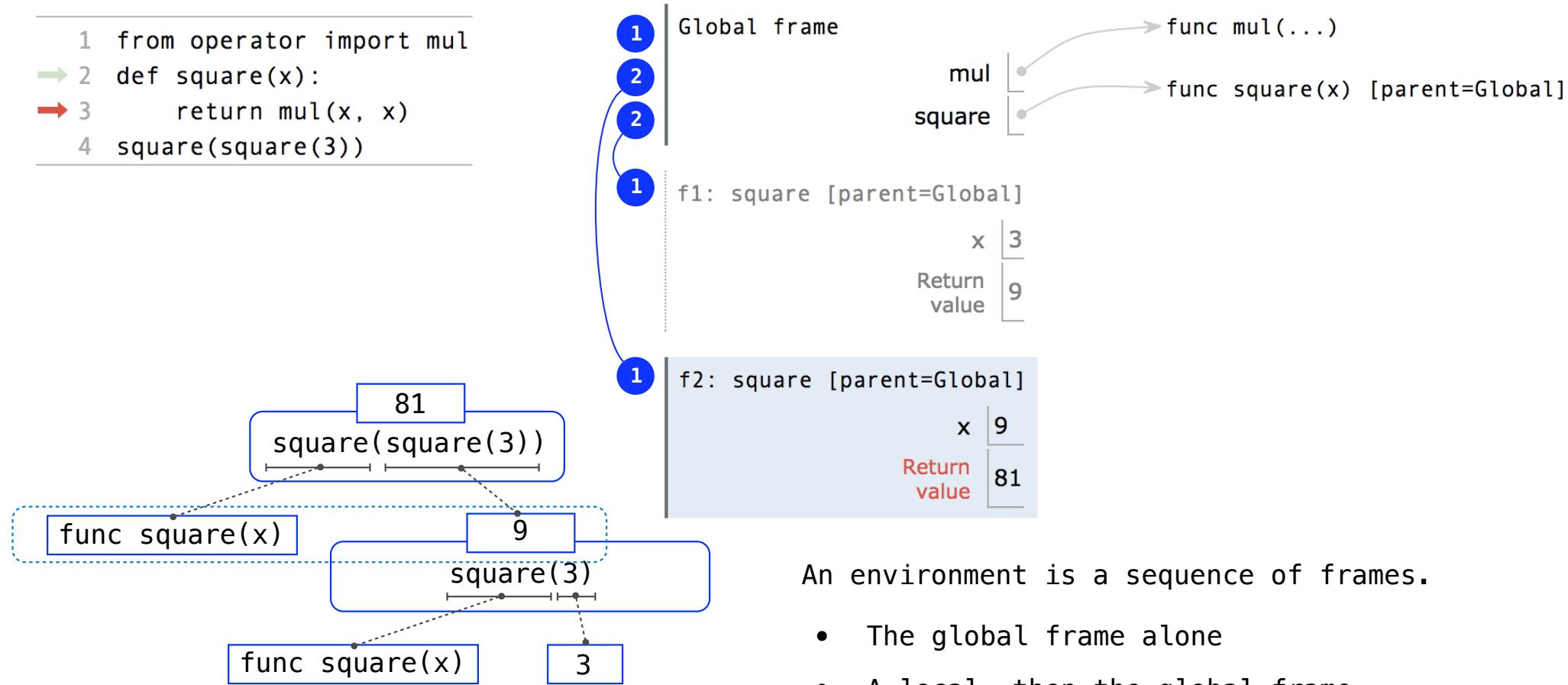
x	3
Return value	9



Interactive Diagram

Multiple Environments in One Diagram!

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



An environment is a sequence of frames.

- The global frame alone
- A local, then the global frame

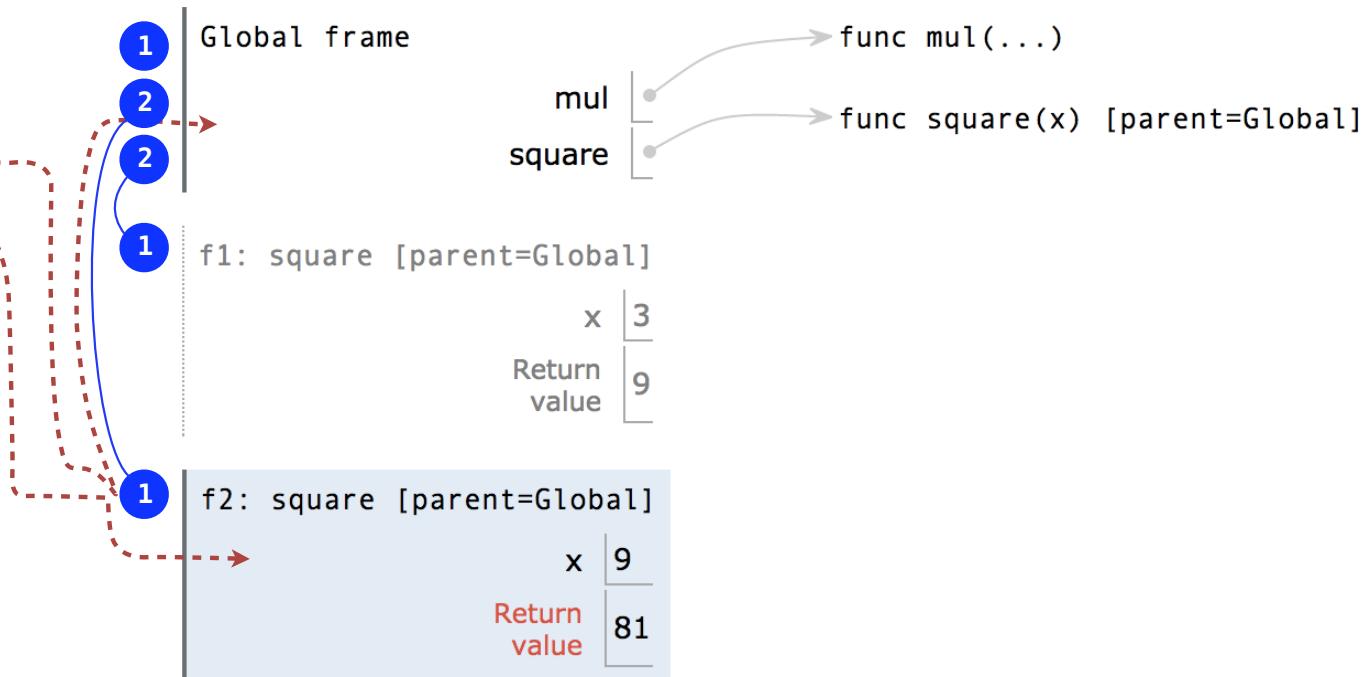
[Interactive Diagram](#)

Names Have No Meaning Without Environments

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

Every expression is evaluated in the context of an environment.

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



An environment is a sequence of frames.

- The global frame alone
- A local, then the global frame

[Interactive Diagram](#)

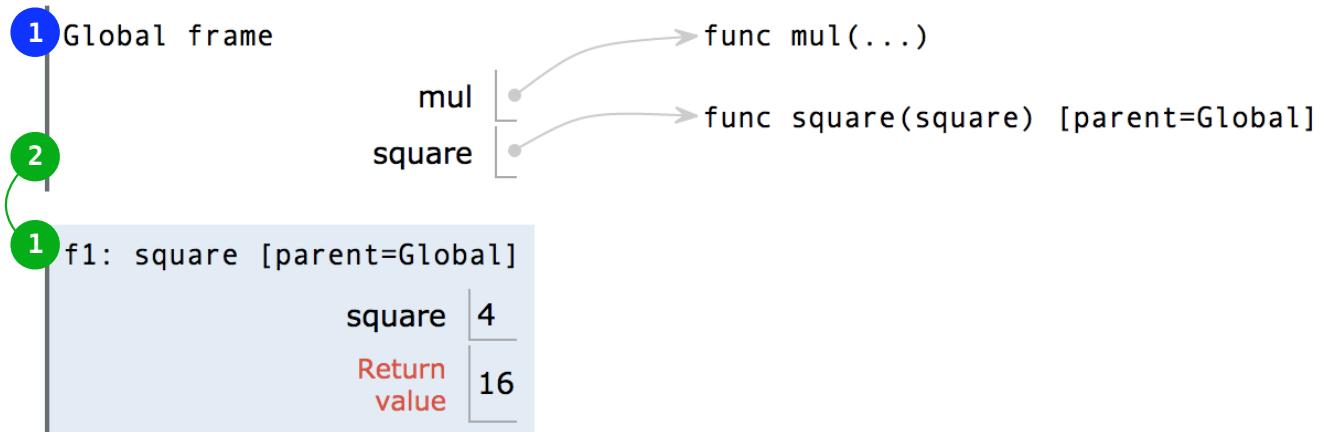
Names Have Different Meanings in Different Environments

A call expression and the body of the function being called
are evaluated in different environments

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



Every expression is
evaluated in the context
of an environment.



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

[Interactive Diagram](#)

13

Environments for Higher-Order Functions

Environments Enable Higher-Order Functions

Functions are first-class: Functions are values in our programming language

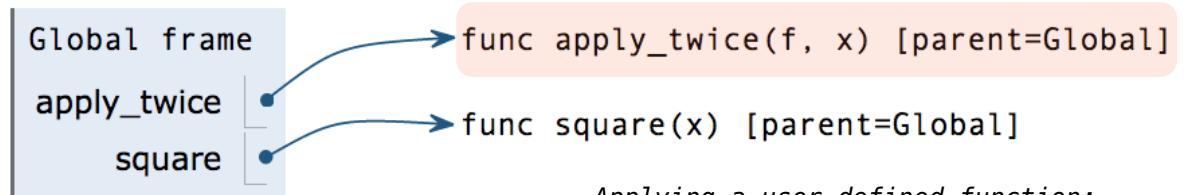
Higher-order function: A function that takes a function as an argument value **or**
A function that returns a function as a return value

Environment diagrams describe how higher-order functions work!

(Demo)

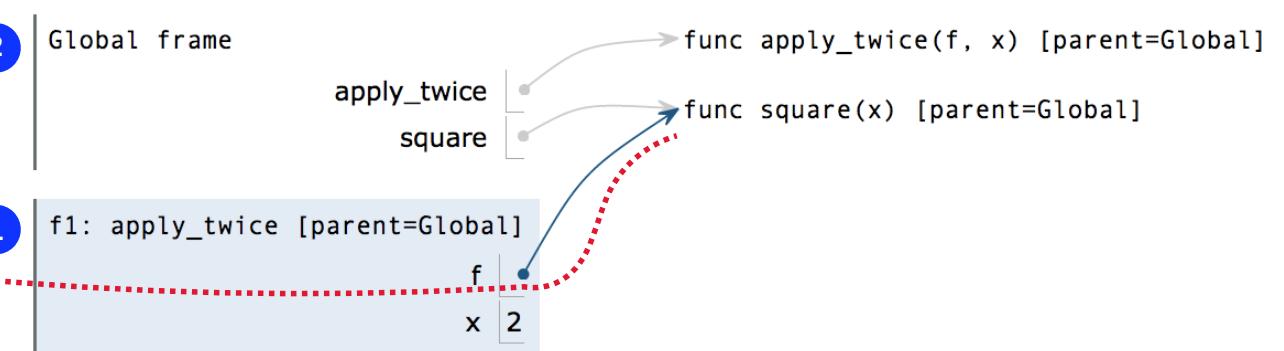
Names can be Bound to Functional Arguments

```
1 def apply_twice(f, x):
2     return f(f(x))
3
4 def square(x):
5     return x * x
6
7 result = apply_twice(square, 2)
```



- Create a new frame
 - Bind formal parameters (f & x) to arguments
 - Execute the body:
$$\text{return } f(f(x))$$

```
1 def apply_twice(f, x):  
2     return f(f(x))  
3  
4 def square(x):  
5     return x * x  
6  
7 result = apply_twice(square, 2)
```



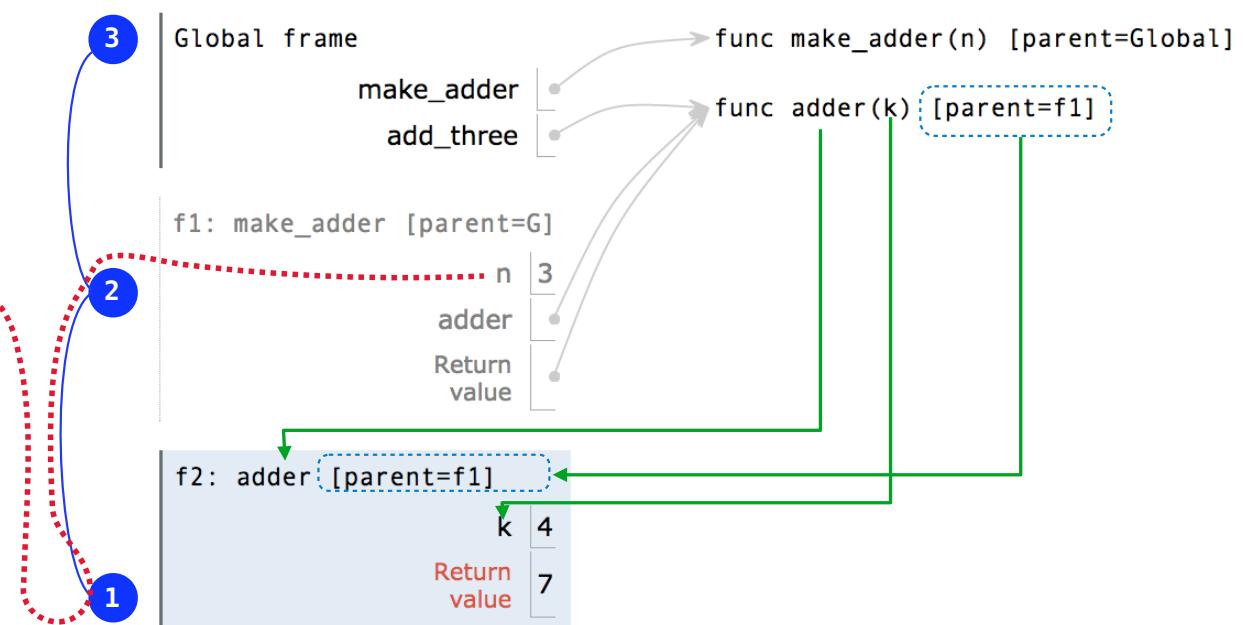
Environments for Nested Definitions

(Demo)

Environment Diagrams for Nested Def Statements

Nested def

```
1 def make_adder(n):
2     def adder(k):
3         return k + n
4     return adder
5
6 add_three = make_adder(3)
7 add_three(4)
```



- Every user-defined function has a parent frame (often global)
 - The parent of a function is the frame in which it was defined
 - Every local frame has a parent frame (often global)
 - The parent of a frame is the parent of the function called

How to Draw an Environment Diagram

When a function is defined:

Create a function value: func <name>(<formal parameters>) [parent=<label>]

Its parent is the current frame.

f1: make_adder

func adder(k) [parent=f1]

Bind <name> to the function value in the current frame

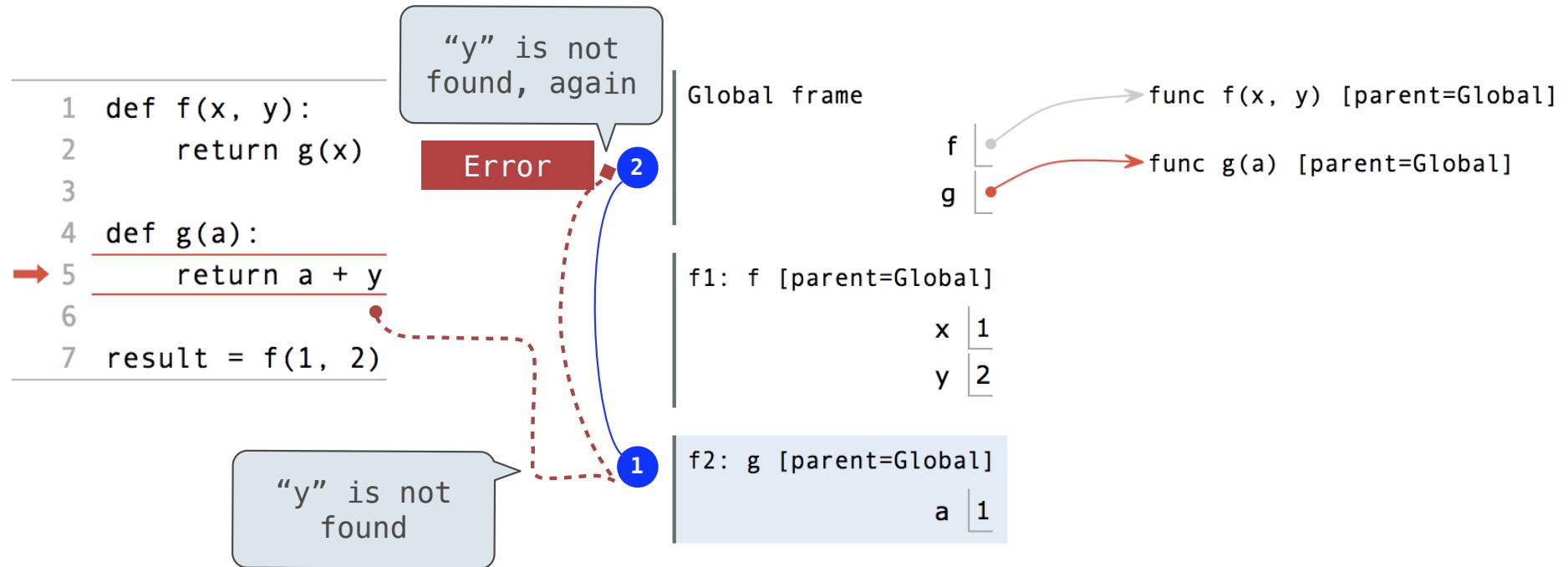
When a function is called:

1. Add a local frame, titled with the <name> of the function being called.
- ★ 2. Copy the parent of the function to the local frame: [parent=<label>]
3. Bind the <formal parameters> to the arguments in the local frame.
4. Execute the body of the function in the environment that starts with the local frame.

Local Names

(Demo)

Local Names are not Visible to Other (Non-Nested) Functions



- An environment is a sequence of frames.
- The environment created by calling a top-level function (no def within def) consists of one local frame, followed by the global frame.

Lambda Expressions

(Demo)

Lambda Expressions

```
>>> x = 10      An expression: this one  
                  evaluates to a number
```



```
>>> square = x * x      Also an expression:  
                           evaluates to a function
```



```
>>> square = lambda x: x * x      Important: No "return" keyword!
```

A function

with formal parameter x

that returns the value of "**x * x**"

```
>>> square(4)      Must be a single expression  
16
```

Lambda expressions are not common in Python, but important in general

Lambda expressions in Python cannot contain statements at all!

Lambda Expressions Versus Def Statements



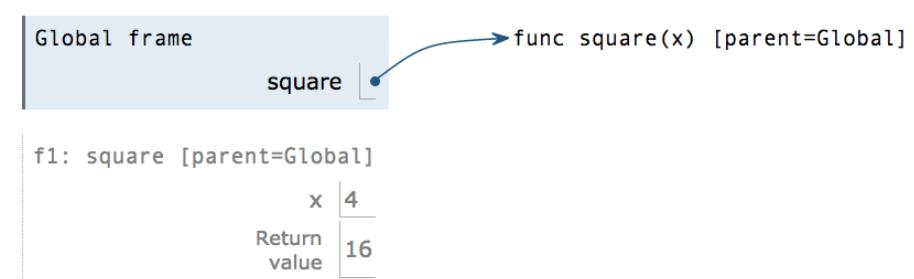
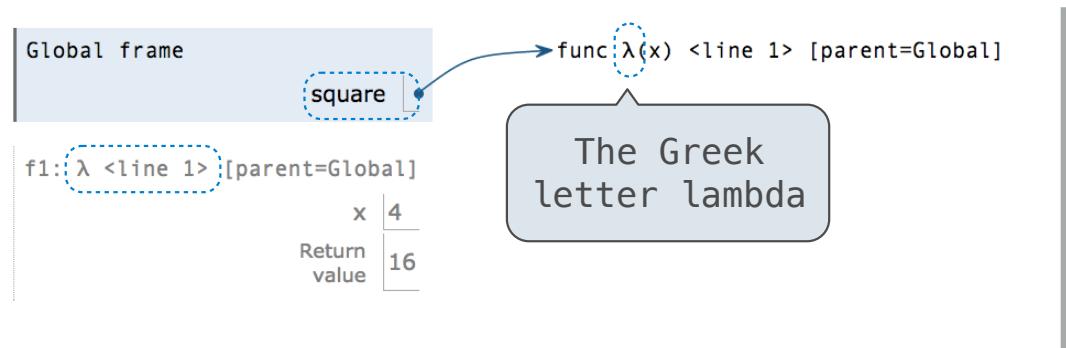
```
square = lambda x: x * x
```

VS

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



- Both create a function with the same domain, range, and behavior.
- Both bind that function to the name square.
- Only the def statement gives the function an intrinsic name, which shows up in environment diagrams but doesn't affect execution (unless the function is printed).

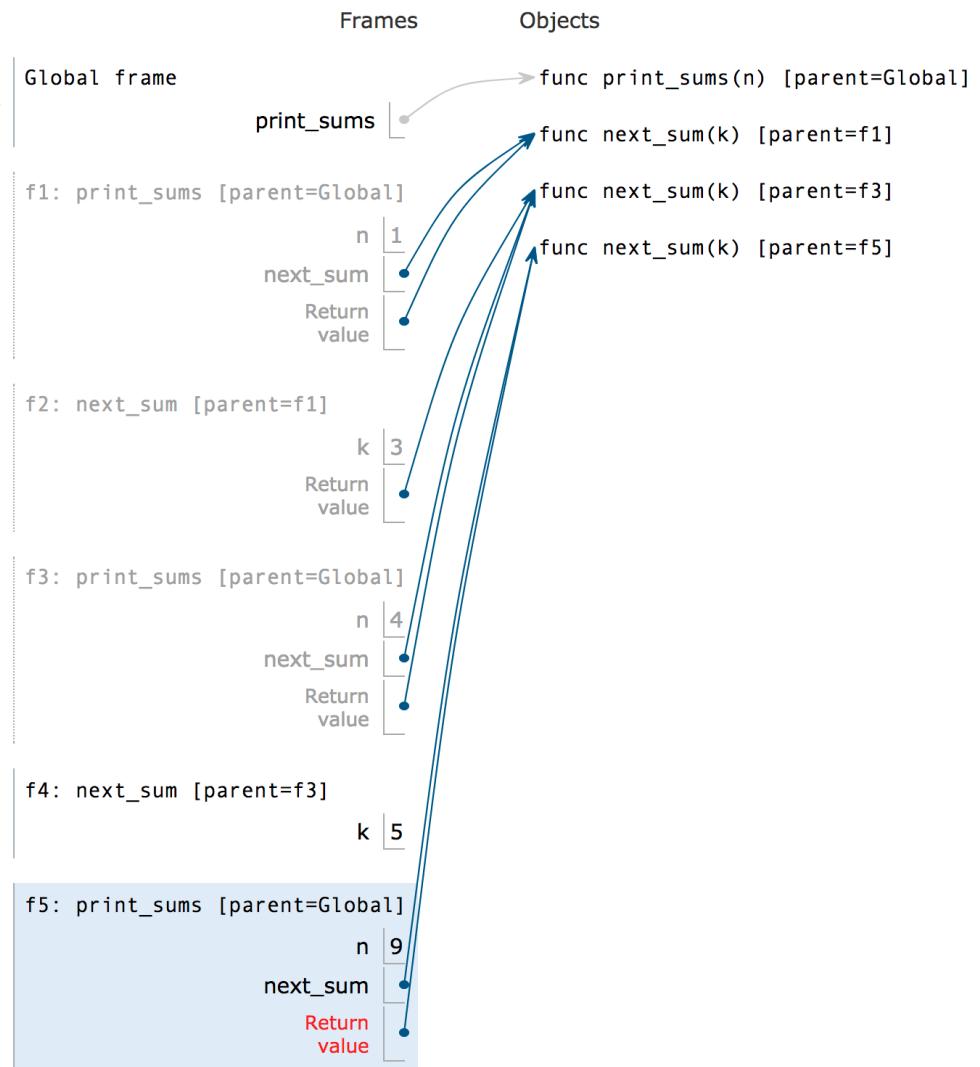


Self-Reference

(Demo)

Returning a Function Using Its Own Name

```
1 def print_sums(n):
2     print(n)
3     def next_sum(k):
4         return print_sums(n+k)
5     return next_sum
6
7 print_sums(1)(3)(5)
```



[http://pythontutor.com/composingprograms.html#code=def%20print_all%28k%29%3A%0A%20%20%20%20print%28k%29%0A%20%20%20%20%20%20def%20next_sum%28k%29%3A%0A%20%20%20%20%20%20return%20print_sums%28n%29%29%0A%20%20%20%20%20%20return%20next_sum%28n%29%29%0A%20%20%20%20%20%20return%20print_sums%28n%29%29%29%285%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D](http://pythontutor.com/composingprograms.html#code=def%20print_all%28k%29%3A%0A%20%20%20%20print%28k%29%0A%20%20%20%20%20%20return%20print_all%281%29%283%29%285%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D)

http://pythontutor.com/composingprograms.html#code=def%20print_sums%28n%29%3A%0A%20%20%20%20print%28n%29%0A%20%20%20%20def%20next_sum%28n%29%3A%0A%20%20%20%20%20%20return%20print_sums%28n%29%29%0A%20%20%20%20%20%20return%20next_sum%28n%29%29%0A%20%20%20%20%20%20return%20print_sums%28n%29%29%29%285%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

Review

What Would Python Display?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

A function that takes any argument and returns a function that returns that arg

```
def delay(arg):
    print('delayed')
    def g():
        return arg
    return g
```

Names in nested def statements can refer to their enclosing scope

This expression	Evaluates to	Interactive Output
5	5	5
print(5)	None	5
print(<u>print(5)</u>)	None	5
None		None
<u>delay(delay))()</u> (6)()	6	delayed delayed 6
print(delay(<u>print()</u> ((4)))	None	delayed 4 None

What Would Python Print?

The `print` function returns `None`. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

A function that
always returns the
identity function

```
def pirate(arggg):
    print('matey')
    def plunder(arggg):
        return arggg
    return plunder
```

This expression	Evaluates to	Interactive Output
<u>add(pirate(3)(square)(4), 1)</u>	17	Matey 17
<u>func square(x)</u>		
<u>16</u>		
<u>pirate(pirate(pirate))(5)(7)</u>	Error	Matey Matey Error
<u>Identity function</u>		
<u>5</u>		

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

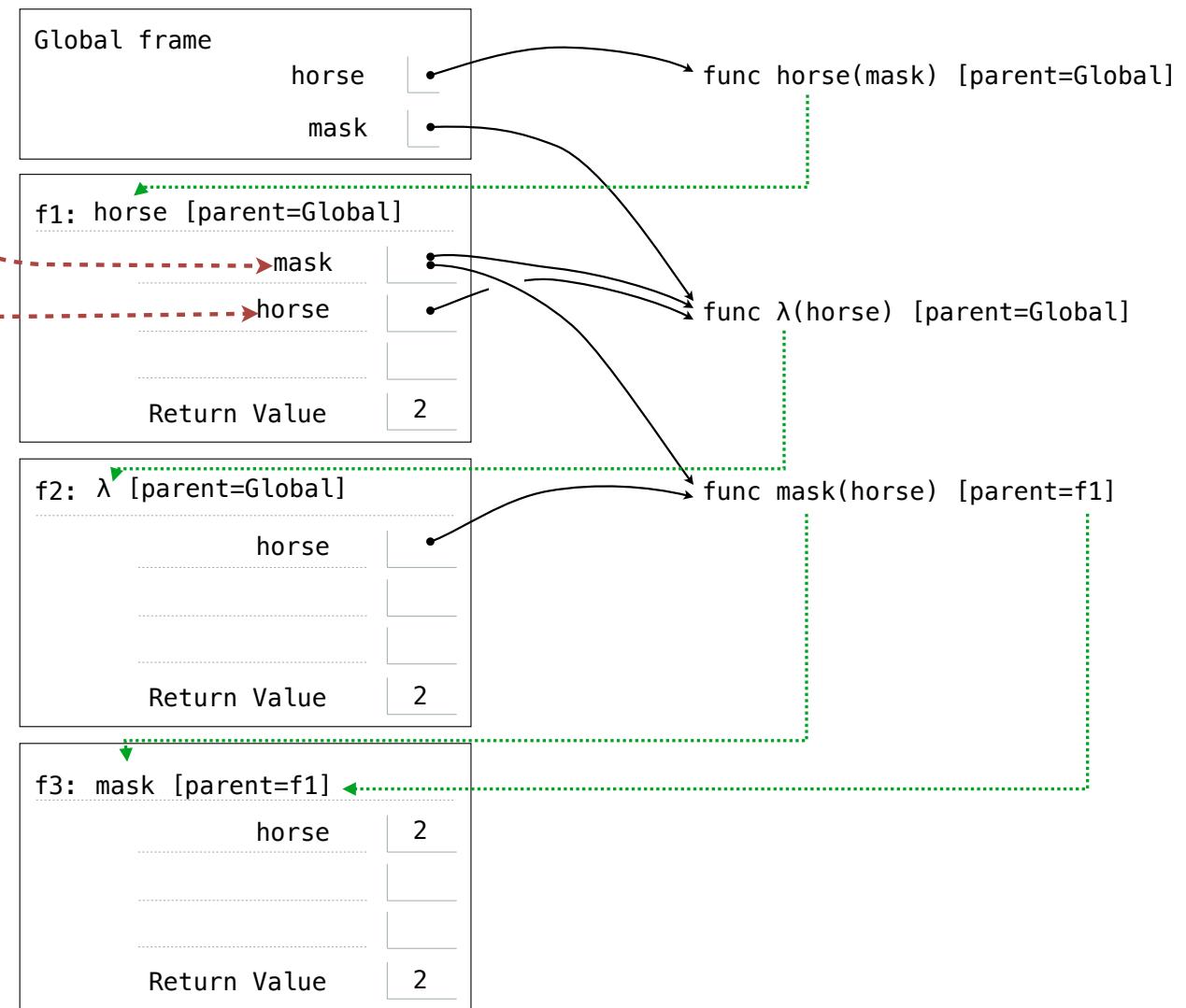
```

def horse(mask):
    horse = mask
def mask(horse):
    return horse
return horse(mask)

mask = lambda horse: horse(2)

horse(mask)

```



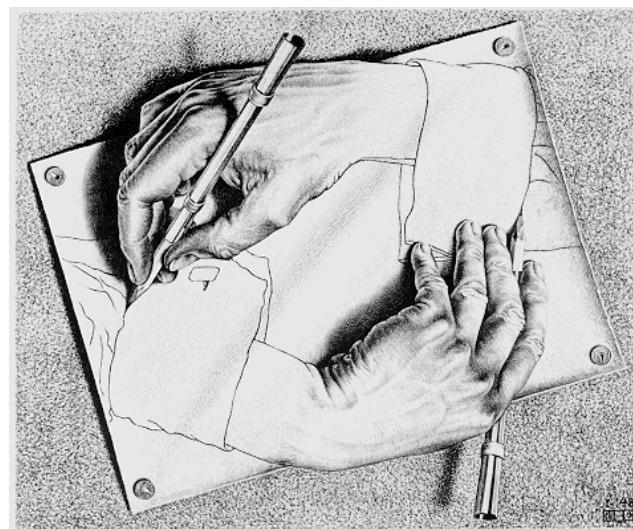
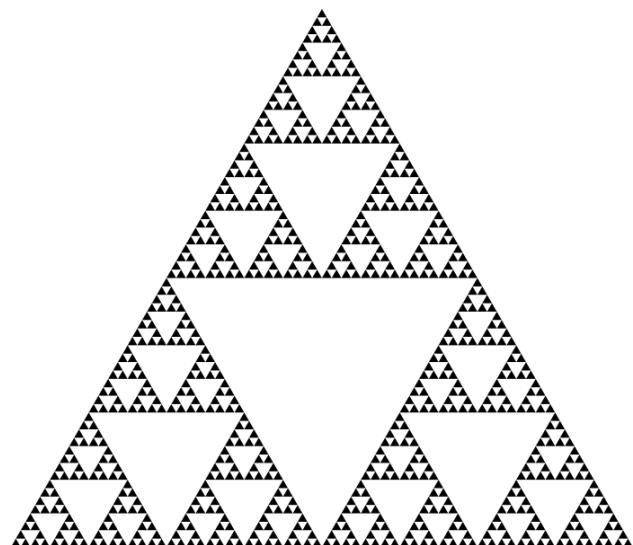
Recursion

Recursive Functions

Recursive Functions

Definition: A function is called recursive if the body of that function calls itself, either directly or indirectly

Implication: Executing the body of a recursive function may require applying that function

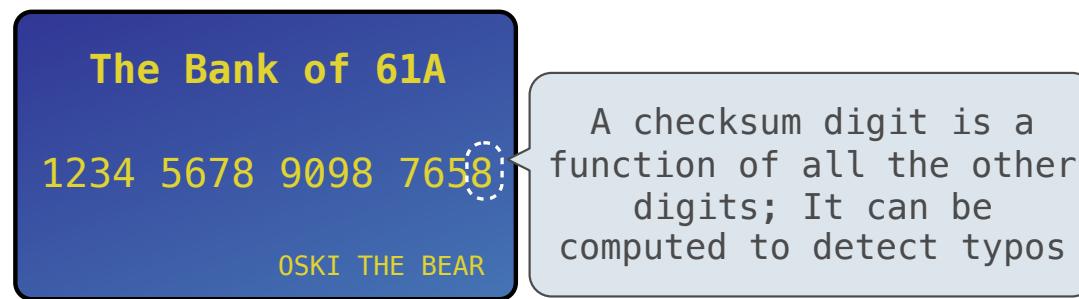


Drawing Hands, by M. C. Escher (lithograph, 1948)

Digit Sums

$$2+0+1+9 = 12$$

- If a number a is divisible by 9, then `sum_digits(a)` is also divisible by 9
- Useful for typo detection!



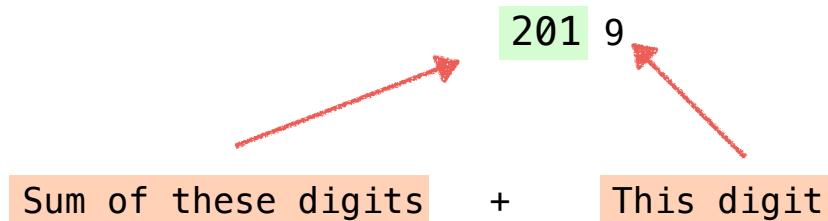
- Credit cards actually use the Luhn algorithm, which we'll implement after `sum_digits`

The Problem Within the Problem

The sum of the digits of 6 is 6.

Likewise for any one-digit (non-negative) number (i.e., < 10).

The sum of the digits of 2019 is



That is, we can break the problem of summing the digits of 2019 into a smaller instance of the same problem, plus some extra stuff.

We call this [recursion](#)

Sum Digits Without a While Statement

```
def split(n):
    """Split positive n into all but its last digit and its last digit."""
    return n // 10, n % 10

def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

The Anatomy of a Recursive Function

- The `def` statement header is similar to other functions
- Conditional statements check for base cases
- Base cases are evaluated without recursive calls
- Recursive cases are evaluated with recursive calls

```
def sum_digits(n):  
    """Return the sum of the digits of positive integer n."""  
  
    if n < 10:  
        return n  
  
    else:  
        all_but_last, last = split(n)  
        return sum_digits(all_but_last) + last
```

(Demo)

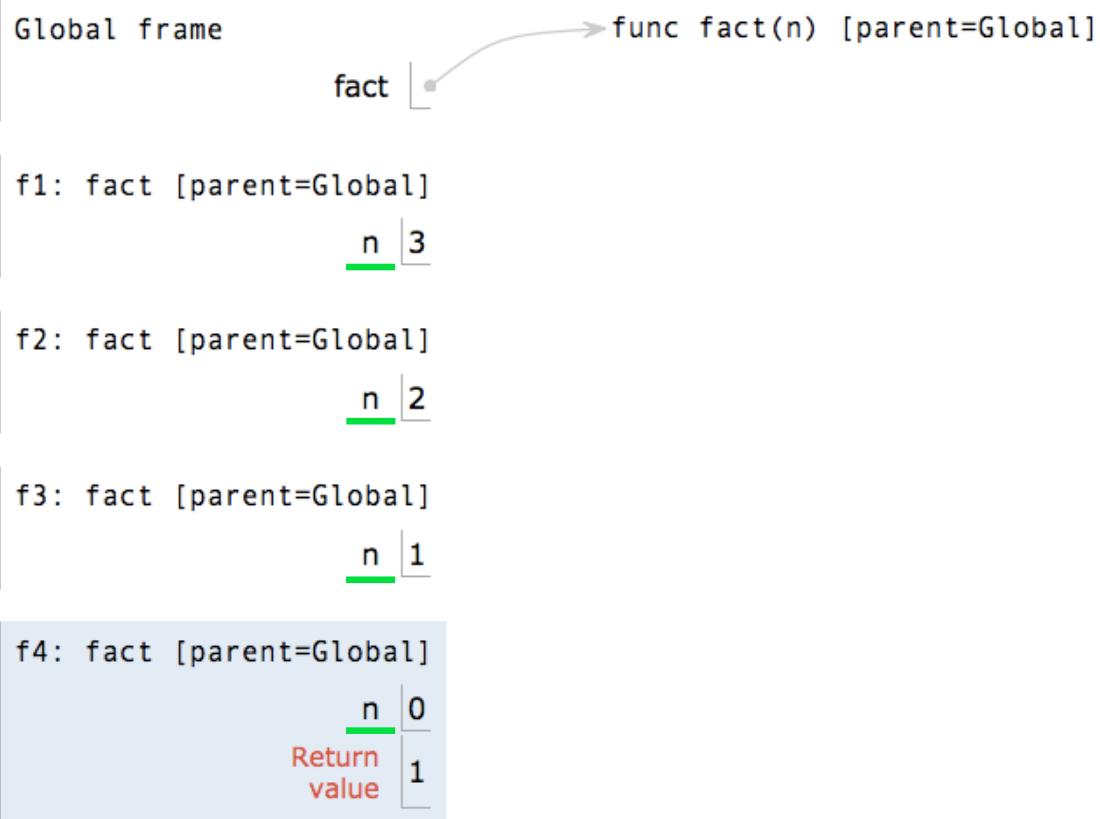
Recursion in Environment Diagrams

Recursion in Environment Diagrams

```
1 def fact(n):
2     if n == 0:
3         return 1
4     else:
5         return n * fact(n-1)
6
7 fact(3)
```

- The same function **fact** is called multiple times
 - Different frames keep track of the different arguments in each call
 - What **n** evaluates to depends upon the current environment
 - Each call to **fact** solves a simpler problem than the last: smaller **n**

(Demo)



Iteration vs Recursion

Iteration is a special case of recursion

$$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$$

Using while:

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

Math:

$$n! = \prod_{k=1}^n k$$

Names:

n, total, k, fact_iter

Using recursion:

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{otherwise} \end{cases}$$

n, fact

Verifying Recursive Functions

The Recursive Leap of Faith

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

Is fact implemented correctly?

1. Verify the base case
2. Treat **fact** as a functional abstraction!
3. Assume that **fact(n-1)** is correct
4. Verify that **fact(n)** is correct

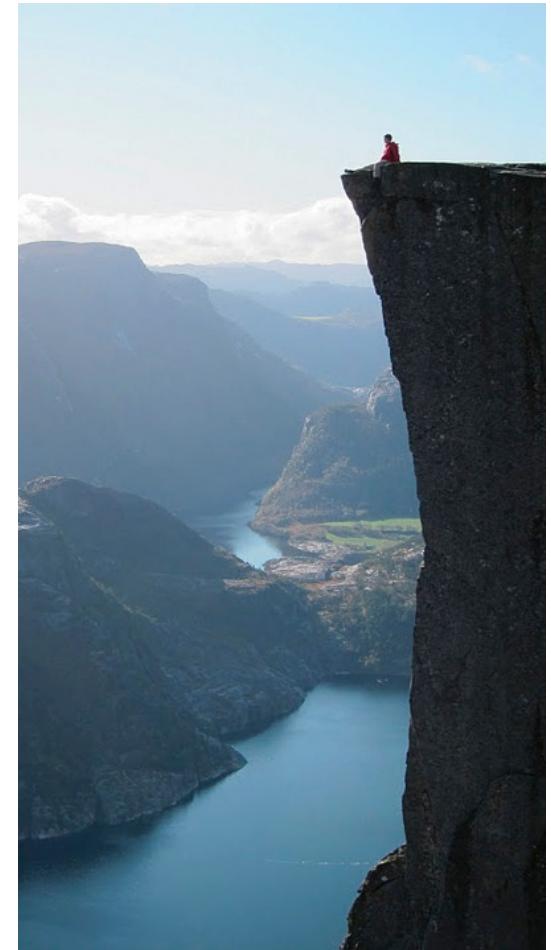


Photo by Kevin Lee, Preikestolen, Norway

Mutual Recursion

The Luhn Algorithm

Used to verify credit card numbers

From Wikipedia: http://en.wikipedia.org/wiki/Luhn_algorithm

- **First:** From the rightmost digit, which is the check digit, moving left, double the value of every second digit; if product of this doubling operation is greater than 9 (e.g., $7 * 2 = 14$), then sum the digits of the products (e.g., 10: $1 + 0 = 1$, 14: $1 + 4 = 5$)
- **Second:** Take the sum of all the digits

1	3	8	7	4	3
2	3	$1+6=7$	7	8	3

$= 30$

The Luhn sum of a valid credit card number is a multiple of 10

(Demo)

Recursion and Iteration

Converting Recursion to Iteration

Can be tricky: Iteration is a special case of recursion.

Idea: Figure out what state must be maintained by the iterative function.

```
def sum_digits(n):  
    """Return the sum of the digits of positive integer n."""  
  
    if n < 10:  
  
        return n  
  
    else:  
  
        all_but_last, last = split(n)
```

```
        return sum_digits(all_but_last) + last
```

What's left to sum

A partial sum

(Demo)

Converting Iteration to Recursion

More formulaic: Iteration is a special case of recursion.

Idea: The state of an iteration can be passed as arguments.

```
def sum_digits_iter(n):
    digit_sum = 0
    while n > 0:
        n, last = split(n)
        digit_sum = digit_sum + last
    return digit_sum
```

Updates via assignment become...

```
def sum_digits_rec(n, digit_sum):
    if n == 0:
        return digit_sum
    else:
        n, last = split(n)
        return sum_digits_rec(n, digit_sum + last)
```

...arguments to a recursive call

Order of Recursive Calls

The Cascade Function

```
1 def cascade(n):
2     if n < 10:
3         print(n)
4     else:
5         print(n)
6         cascade(n//10)
7         print(n)
8
9 cascade(123)
```

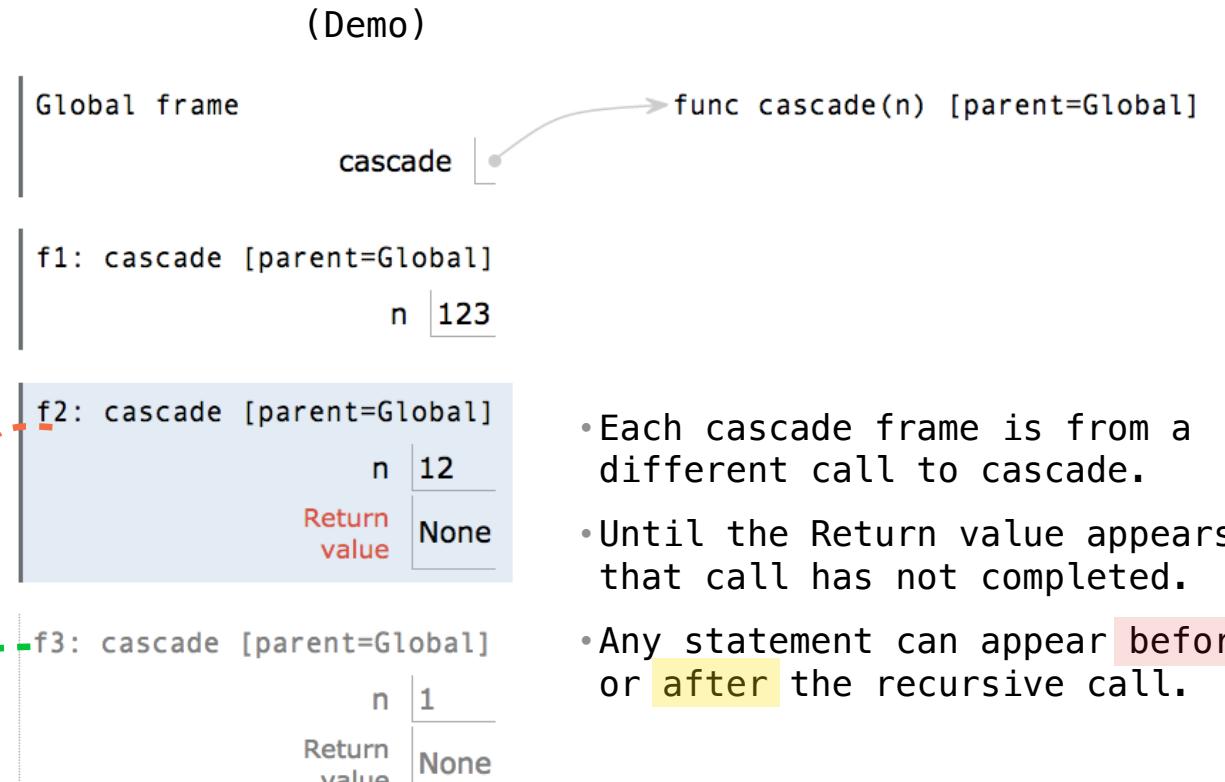
Program output:

123

12 

1

12



- Each cascade frame is from a different call to cascade.
 - Until the Return value appears, that call has not completed.
 - Any statement can appear before or after the recursive call.

Two Definitions of Cascade

(Demo)

```
def cascade(n):
    if n < 10:
        print(n)
    else:
        print(n)
        cascade(n//10)
        print(n)
```

```
def cascade(n):
    print(n)
    if n >= 10:
        cascade(n//10)
        print(n)
```

- If two implementations are equally clear, then shorter is usually better
- In this case, the longer implementation is more clear (at least to me)
- When learning to write recursive functions, put the base cases first
- Both are recursive functions, even though only the first has typical structure

Example: Inverse Cascade

Inverse Cascade

Write a function that prints an inverse cascade:

```
1           def inverse_cascade(n):
12          grow(n)
123         print(n)
1234        shrink(n)
123
123           def f_then_g(f, g, n):
12           if n:
1
f(n)
g(n)

grow = lambda n: f_then_g( )
shrink = lambda n: f_then_g( )
```

Tree Recursion

Tree Recursion

Tree-shaped processes arise whenever executing the body of a recursive function makes more than one recursive call

`n: 0, 1, 2, 3, 4, 5, 6, 7, 8, ... , 35`

`fib(n): 0, 1, 1, 2, 3, 5, 8, 13, 21, ... , 9,227,465`

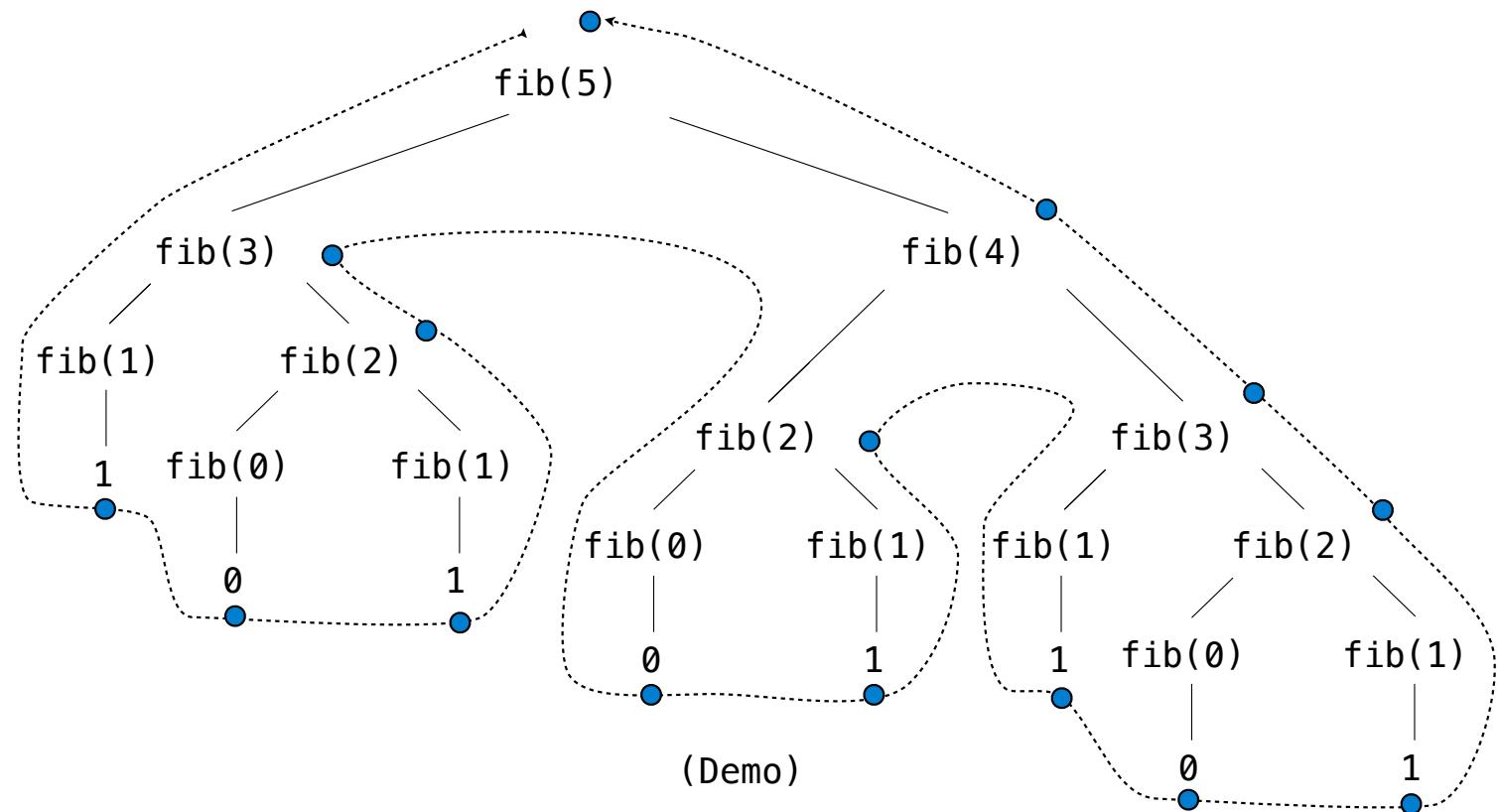
```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```



<http://en.wikipedia.org/wiki/File:Fibonacci.jpg>

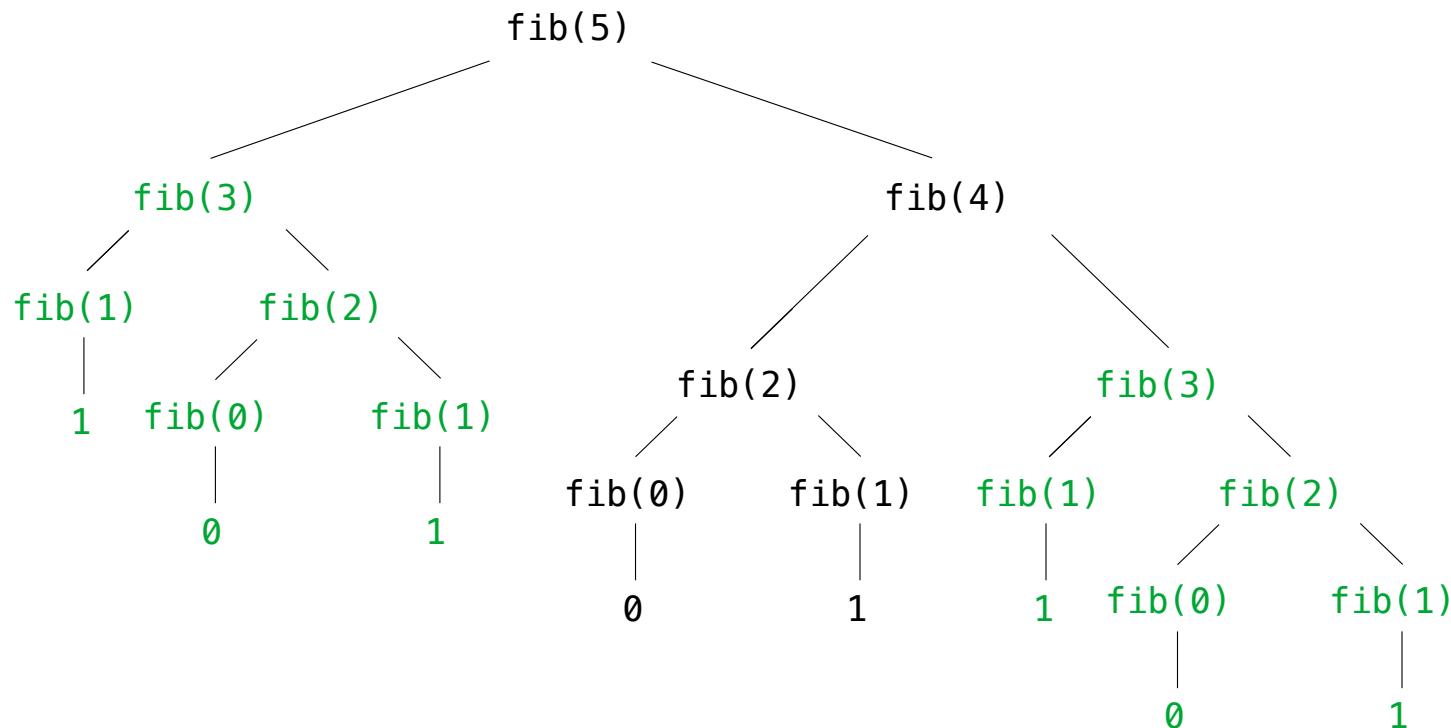
A Tree-Recursive Process

The computational process of fib evolves into a tree structure



Repetition in Tree-Recursive Computation

This process is highly repetitive; fib is called on the same argument multiple times



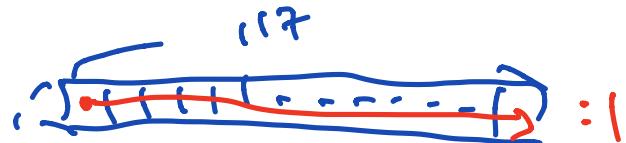
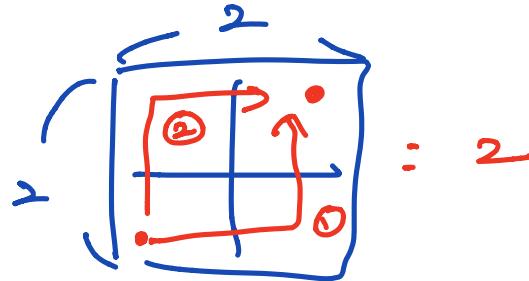
(We will speed up this computation dramatically in a few weeks by remembering results)

```
def path (m, n):
    """Return the number of paths from one corner of an
    M by N grid to the opposite corner.
```

```
>>> path (2, 2)
2
>>> path (5, 7)
210
>>> path (117, 117)
1
>>> path (1, 157)
1
....
```

```
if m==1 or n==1:
    return 1
```

```
return path(m-1, n) + path(m, n-1)
```



Total # of ways to get to (M, N)

= Total # of ways to get to $(M-1, N)$ +
 Total # of ways to get to $(M, N-1)$

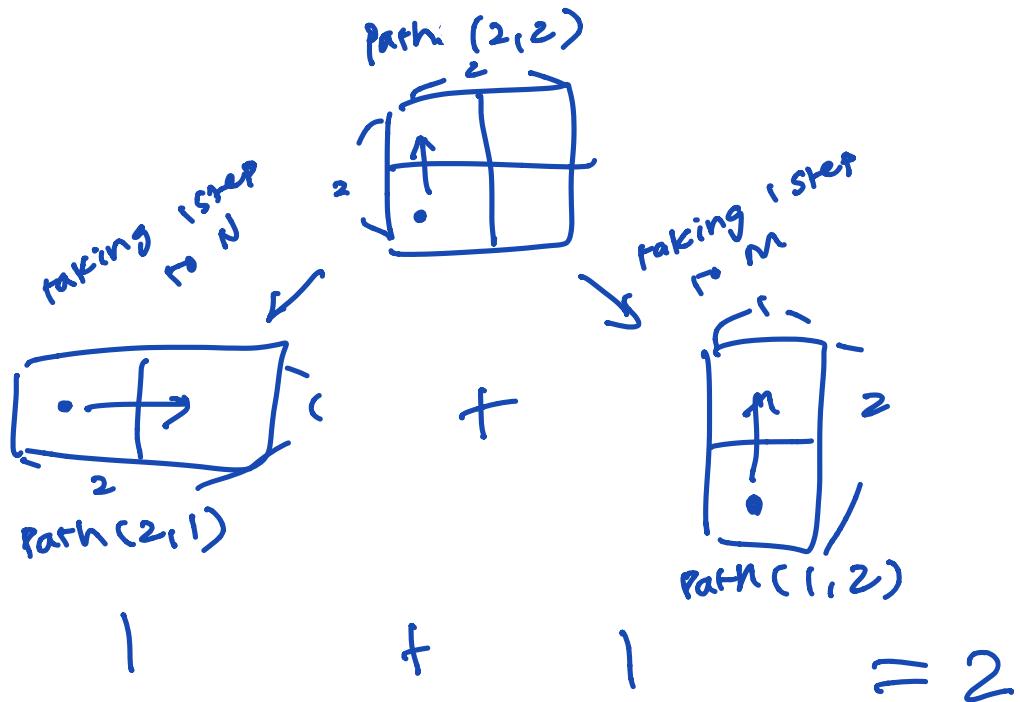
→ took 1 step towards M

→ took 1 step towards N

Total # of ways to get to (M, N)

= Total # of ways to get to $(M-1, N)$ +

Total # of ways to get to $(M, N-1)$



$n = 689$ $k = 16$

$n // 10$ $n \% 10$



```
def knap(n, k):  
    if n == 0:  
        return k == 0  
  
    with_last = knap(n // 10, k - n % 10)  
    without_last = knap(n // 10, k)  
  
    return with_last or without_last
```

Example: Counting Partitions

Counting Partitions

The number of partitions of a positive integer n , using parts up to size m , is the number of ways in which n can be expressed as the sum of positive integer parts up to m in increasing order.

count_partitions(6, 4)

$$2 + 4 = 6$$



$$1 + 1 + 4 = 6$$



$$3 + 3 = 6$$



$$1 + 2 + 3 = 6$$



$$1 + 1 + 1 + 3 = 6$$



$$2 + 2 + 2 = 6$$



$$1 + 1 + 2 + 2 = 6$$



$$1 + 1 + 1 + 1 + 2 = 6$$



$$1 + 1 + 1 + 1 + 1 + 1 + 1 = 6$$

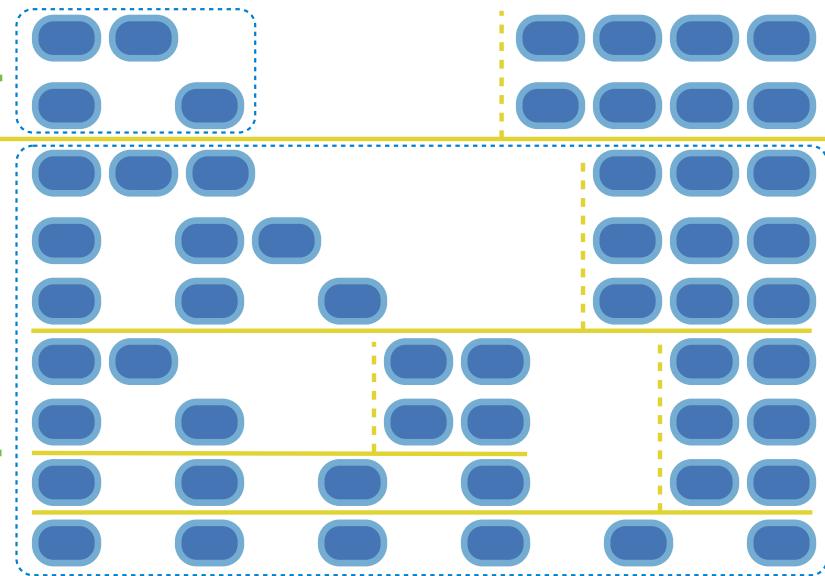


Counting Partitions

The number of partitions of a positive integer n , using parts up to size m , is the number of ways in which n can be expressed as the sum of positive integer parts up to m in non-decreasing order.

`count_partitions(6, 4)`

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
 - Use at least one 4
 - Don't use any 4
- Solve two simpler problems:
 - `count_partitions(2, 4)`
 - `count_partitions(6, 3)`
- Tree recursion often involves exploring different choices.



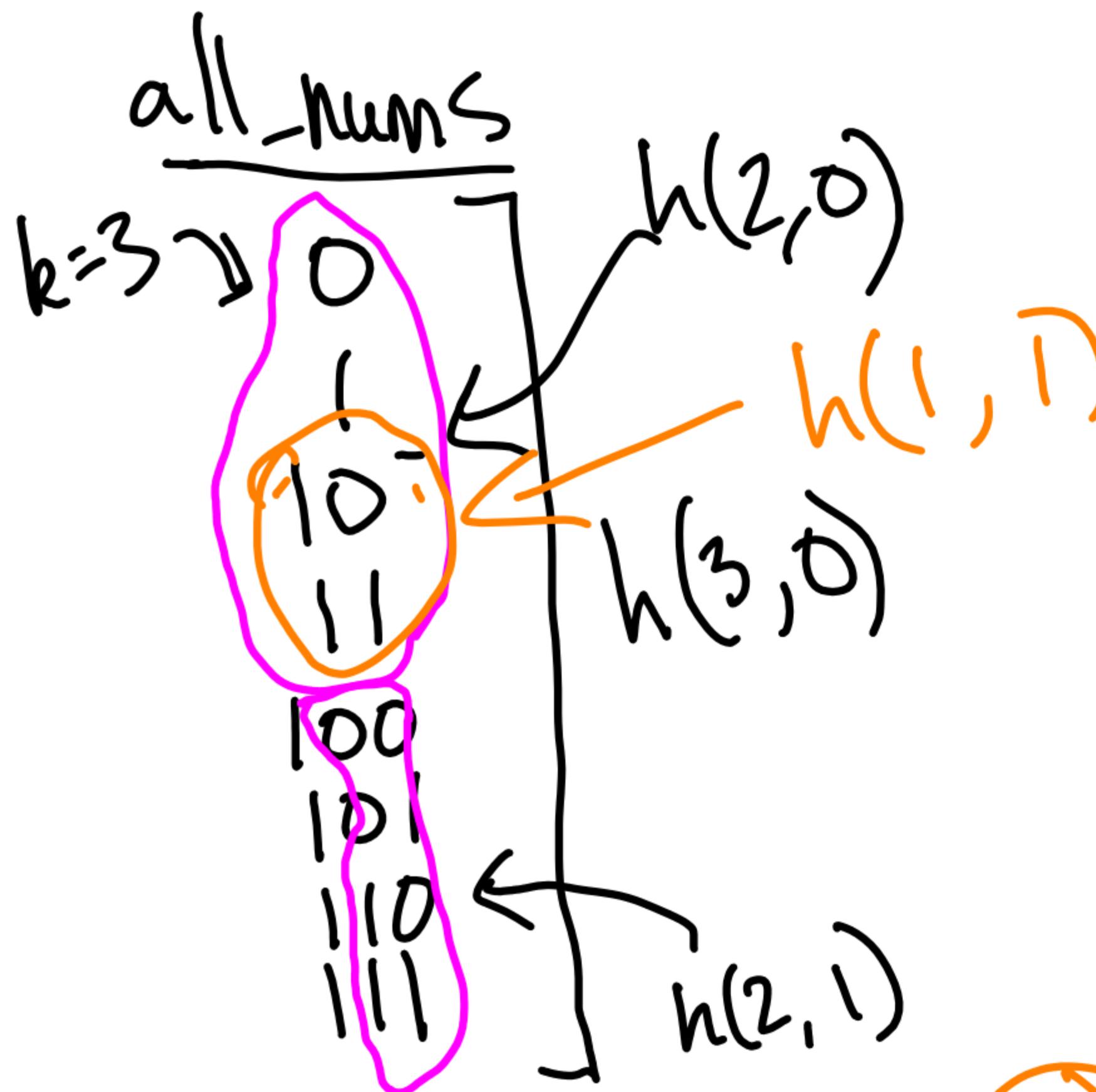
Counting Partitions

The number of partitions of a positive integer n , using parts up to size m , is the number of ways in which n can be expressed as the sum of positive integer parts up to m in increasing order.

- Recursive decomposition: finding simpler instances of the problem.
 - Explore two possibilities:
 - Use at least one 4
 - Don't use any 4
 - Solve two simpler problems:
 - `count_partitions(2, 4)` -----
 - `count_partitions(6, 3)` -----
 - Tree recursion often involves exploring different choices.

```
def count_partitions(n, m):
    if n == 0:
        return 1
    elif n < 0:
        return 0
    elif m == 0:
        return 0
    else:
        with_m = count_partitions(n-m, m)
        without_m = count_partitions(n, m-1)
        return with_m + without_m
```

(Demo)



```

def all_nums(k):
    def h(k, prefix):
        if k == 0:
            print(prefix)
            return
        - h(k-1, prefix + "0")
        - h(k-1, prefix + "1")
    h(k, 0)

```



Lists

['Demo']

Working with Lists

```
>>> digits = [1, 8, 2, 8]
```

The number of elements

```
>>> len(digits)  
4
```

An element selected by its index

```
>>> digits[3]  
8
```

Concatenation and repetition

```
>>> [2, 7] + digits * 2  
[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]
```

```
>>> digits = [2//2, 2+2+2+2, 2, 2*2*2]
```

```
>>> getitem(digits, 3)  
8
```

```
>>> add([2, 7], mul(digits, 2))  
[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]
```

Nested lists

```
>>> pairs = [[10, 20], [30, 40]]  
>>> pairs[1]  
[30, 40]  
>>> pairs[1][0]  
30
```

Containers

Containers

Built-in operators for testing whether an element appears in a compound value

```
>>> digits = [1, 8, 2, 8]
>>> 1 in digits
True
>>> 8 in digits
True
>>> 5 not in digits
True
>>> not(5 in digits)
True
```

(Demo)

For Statements

(Demo)

Sequence Iteration

```
def count(s, value):
    total = 0
    for element in s:
        if element == value:
            total = total + 1
    return total
```

Name bound in the first frame
of the current environment
(not a new frame)

For Statement Execution Procedure

```
for <name> in <expression>:  
    <suite>
```

1. Evaluate the header <expression>, which must yield an iterable value (a sequence)
2. For each element in that sequence, in order:
 - A. Bind <name> to that element in the current frame
 - B. Execute the <suite>

Sequence Unpacking in For Statements

A sequence of
fixed-length sequences

```
>>> pairs = [[1, 2], [2, 2], [3, 2], [4, 4]]  
>>> same_count = 0
```

A name for each element in a
fixed-length sequence

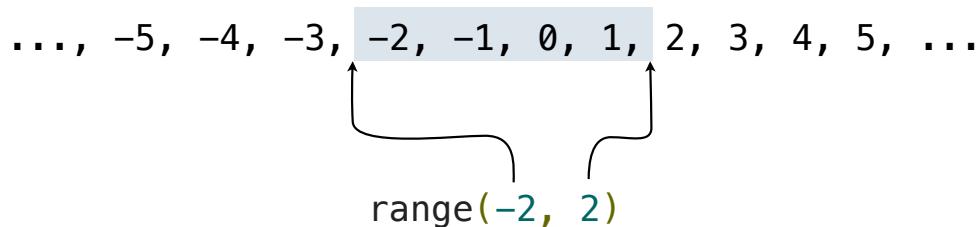
Each name is bound to a value, as in
multiple assignment

```
>>> for x, y in pairs:  
...     if x == y:  
...         same_count = same_count + 1  
  
>>> same_count  
2
```

Ranges

The Range Type

A range is a sequence of consecutive integers.*



Length: ending value – starting value

(Demo)

Element selection: starting value + index

```
>>> list(range(-2, 2))
```

[-2, -1, 0, 1]

List constructor

```
>>> list(range(4))
```

[0, 1, 2, 3]

Range with a 0 starting value

* Ranges can actually represent more general integer sequences.

List Comprehensions

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'm', 'n', 'o', 'p']
>>> [letters[i] for i in [3, 4, 6, 8]]
['d', 'e', 'm', 'o']
```

List Comprehensions

[<map exp> for <name> in <iter exp> if <filter exp>]

Short version: [<map exp> for <name> in <iter exp>]

A combined expression that evaluates to a list using this evaluation procedure:

1. Add a new frame with the current frame as its parent
2. Create an empty *result list* that is the value of the expression
3. For each element in the iterable value of <iter exp>:
 - A. Bind <name> to that element in the new frame from step 1
 - B. If <filter exp> evaluates to a true value, then add the value of <map exp> to the result list

Strings

Strings are an Abstraction

Representing data:

```
'200'      '1.2e-5'      'False'      '[1, 2]'
```

Representing language:

```
"""And, as imagination bodies forth  
The forms of things unknown, the poet's pen  
Turns them to shapes, and gives to airy nothing  
A local habitation and a name.  
"""
```

Representing programs:

```
'curry = lambda f: lambda x: lambda y: f(x, y)'
```

(Demo)

String Literals Have Three Forms

```
>>> 'I am string!'
'I am string!'
```

```
>>> "I've got an apostrophe"
"I've got an apostrophe"
```

```
>>> '您好'
'您好'
```

```
>>> """The Zen of Python
claims, Readability counts.
Read more: import this."""
'The Zen of Python\nclaims, Readability counts.\nRead more: import this.'
```

Single-quoted and double-quoted strings are equivalent

A backslash "escapes" the following character

"Line feed" character represents a new line

Dictionaries

{'Dem': 0}

Limitations on Dictionaries

Dictionaries are **unordered** collections of key-value pairs

Dictionary keys do have two restrictions:

- A key of a dictionary **cannot be** a list or a dictionary (or any *mutable type*)
- Two **keys cannot be equal**; There can be at most one value for a given key

This first restriction is tied to Python's underlying implementation of dictionaries

The second restriction is part of the dictionary abstraction

If you want to associate multiple values with a key, store them all in a sequence value

Data Abstraction

Data Abstraction

- Compound values combine other values together
 - A date: a year, a month, and a day
 - A geographic position: latitude and longitude
- Data abstraction lets us manipulate compound values as units
- Isolate two parts of any program that uses data:
 - How data are represented (as parts)
 - How data are manipulated (as units)
- Data abstraction: A methodology by which functions enforce an abstraction barrier between **representation** and **use**

Rational Numbers

$$\frac{\text{numerator}}{\text{denominator}}$$

Exact representation of fractions

A pair of integers

As soon as division occurs, the exact representation may be lost! (Demo)

Assume we can compose and decompose rational numbers:

Constructor

→ `rational(n, d)` returns a rational number x

Selectors

• `numer(x)` returns the numerator of x

• `denom(x)` returns the denominator of x

Rational Number Arithmetic

$$\frac{3}{2} * \frac{3}{5} = \frac{9}{10}$$

$$\frac{3}{2} + \frac{3}{5} = \frac{21}{10}$$

Example

$$\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx*ny}{dx*dy}$$

$$\frac{nx}{dx} + \frac{ny}{dy} = \frac{nx*dy + ny*dx}{dx*dy}$$

General Form

Rational Number Arithmetic Implementation

```
def mul_rational(x, y):
    return rational(numer(x) * numer(y),
                    denom(x) * denom(y))
```

Constructor

Selectors

```
def add_rational(x, y):
    nx, dx = numer(x), denom(x)
    ny, dy = numer(y), denom(y)
    return rational(nx * dy + ny * dx, dx * dy)
```

```
def print_rational(x):
    print(numer(x), '/', denom(x))
```

```
def rationals_are_equal(x, y):
    return numer(x) * denom(y) == numer(y) * denom(x)
```

- `rational(n, d)` returns a rational number x
- `numer(x)` returns the numerator of x
- `denom(x)` returns the denominator of x

These functions implement an abstract representation for rational numbers

$$\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx*ny}{dx*dy}$$

$$\frac{nx}{dx} + \frac{ny}{dy} = \frac{nx*dy + ny*dx}{dx*dy}$$

Pairs

Representing Pairs Using Lists

```
>>> pair = [1, 2]          A list literal:  
>>> pair  
[1, 2]                    Comma-separated expressions in brackets  
  
>>> x, y = pair          "Unpacking" a list  
>>> x  
1  
>>> y  
2  
  
>>> pair[0]               Element selection using the selection operator  
1  
>>> pair[1]  
2  
  
>>> from operator import getitem    Element selection function  
>>> getitem(pair, 0)  
1  
>>> getitem(pair, 1)  
2
```

Representing Rational Numbers

```
def rational(n, d):
    """Construct a rational number that represents N/D."""
    return [n, d]
```

Construct a list

```
def numer(x):
    """Return the numerator of rational number X."""
    return x[0]
```

```
def denom(x):
    """Return the denominator of rational number X."""
    return x[1]
```

Select item from a list

(Demo)

Reducing to Lowest Terms

Example:

$$\frac{3}{2} * \frac{5}{3} = \frac{5}{2}$$

$$\frac{2}{5} + \frac{1}{10} = \frac{1}{2}$$

$$\frac{15}{6} * \frac{1/3}{1/3} = \frac{5}{2}$$

$$\frac{25}{50} * \frac{1/25}{1/25} = \frac{1}{2}$$

```
from fractions import gcd
def rational(n, d):
    """Construct a rational that represents n/d in lowest terms."""
    g = gcd(n, d)
    return [n//g, d//g]
```

(Demo)

Abstraction Barriers

Abstraction Barriers

Parts of the program that...	Treat rationals as...	Using...
Use rational numbers to perform computation	whole data values	<code>add_rational, mul_rational rationals_are_equal, print_rational</code>
Create rationals or implement rational operations	numerators and denominators	<code>rational, numer, denom</code>
Implement selectors and constructor for rationals	two-element lists	list literals and element selection

Implementation of lists

Violating Abstraction Barriers

```
Does not use  
constructors
```

```
Twice!
```

```
add_rational( [1, 2], [1, 4] )
```

```
def divide_rational(x, y):  
    return [x[0] * y[1], x[1] * y[0]]
```

```
No selectors!
```

```
And no constructor!
```

Data Representations

What are Data?

- We need to guarantee that constructor and selector functions work together to specify the right behavior
- Behavior condition: If we construct rational number x from numerator n and denominator d , then $\text{numer}(x)/\text{denom}(x)$ must equal n/d
- Data abstraction uses selectors and constructors to define behavior
- If behavior conditions are met, then the representation is valid

You can recognize an abstract data representation by its behavior

(Demo)

Rationals Implemented as Functions

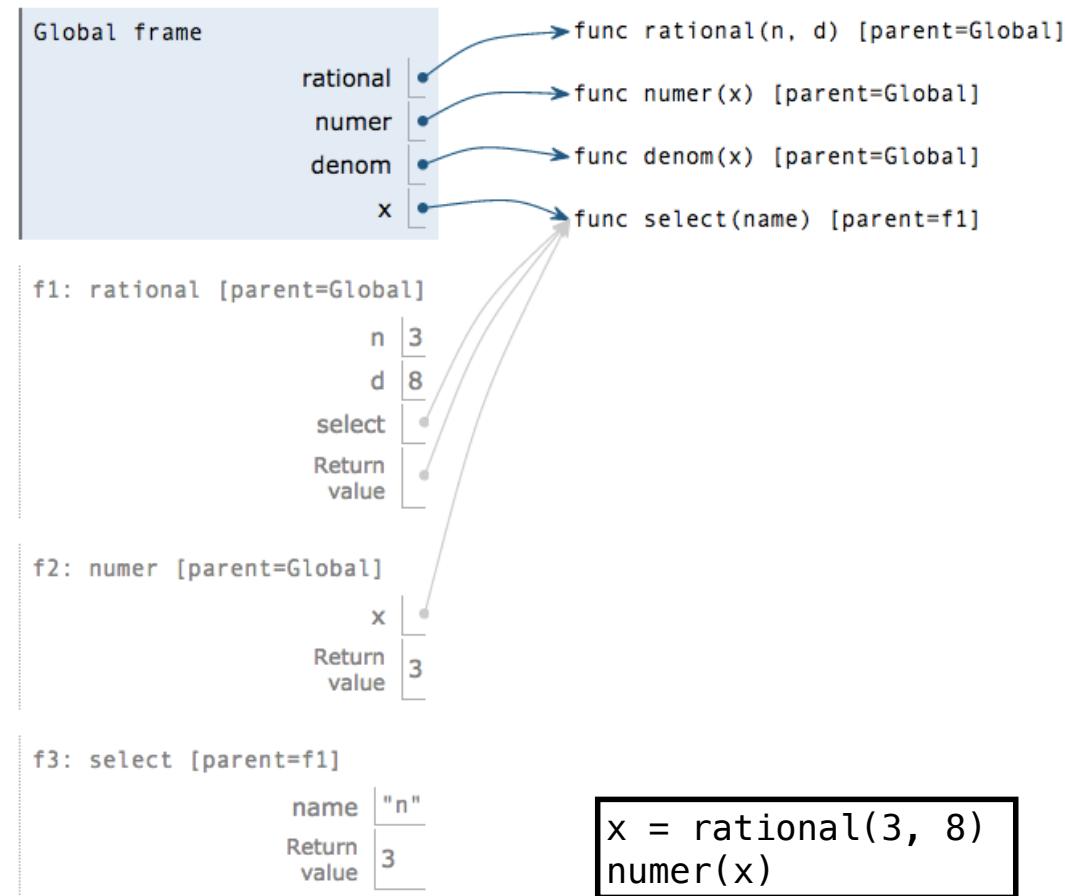
```
def rational(n, d):
    def select(name):
        if name == 'n':
            return n
        elif name == 'd':
            return d
    return select
```

```
def numer(x):  
    return x('n')  
  
def denom(x):  
    return x('d')
```

This function represents a rational number

Constructor is a higher-order function

Selector calls x



Dictionaries

{'Dem': 0}

Limitations on Dictionaries

Dictionaries are **unordered** collections of key-value pairs

Dictionary keys do have two restrictions:

- A key of a dictionary **cannot be** a list or a dictionary (or any *mutable type*)
- Two **keys cannot be equal**; There can be at most one value for a given key

This first restriction is tied to Python's underlying implementation of dictionaries

The second restriction is part of the dictionary abstraction

If you want to associate multiple values with a key, store them all in a sequence value

Debugging

“Beware of bugs in the above code; I have only proved it correct, not tried it.”
-David Knuth

assert

Assertions: Use

- What happens if you run half_fact(5)?
 - Infinite loop??????
- Code should fail as soon as possible
 - Makes error detection easier
- Assertions are forever

```
def fact(x):  
    assert isinstance(x, int)  
    assert x >= 0  
    if x == 0:  
        return 1  
    else:  
        return x * fact(x - 1)
```

```
def half_fact(x):  
    return fact(x / 2)
```

Assertions: Limitations

- Require invariants
 - Assertions tend to be useful when you know a good invariant
 - An **invariant** is something that is always true
 - E.g., the argument to fact being a non-negative integer
- Assertions *check that code meets an existing understanding*
 - They are less useful at actually developing an understanding of how some code is working
 - Generally, assertions are best added to your *own* code, not someone else's
 - (For the purpose of debugging, you six months ago is a different person)

Assertions: Limitations demo

- What assertion should be added here?

```
def t(f, n, x, x0=0):  
    assert ???  
    r = 0  
    while n:  
        r += (x-x0) ** n / fact(n) * d(n, f)(x0)  
        n -= 1  
    return r
```

Testing

Testing: Why do it?

- *Detect errors in your code*
- *Have confidence in the correctness of subcomponents*
- *Narrow down the scope of debugging*
- *Document how your code works*

Testing: Doctests

- Python provides a way to write tests as part of the docstring
- Just put the arrows and go!
- Right there with the code and docs
- To run:
 - python3 -m doctest file.py

```
# in file.py
def fib(n):
    """Fibonacci
    >>> fib(2)
    1
    >>> fib(10)
    55
    """
    ...
    ...
```

Testing: Doctest Limitations

- Doctests have to be in the file
 - Can't be too many
- Do not treat print/return differently
 - Makes print debugging difficult
 - ok fixes this issue

```
def fib(n):  
    """Fibonacci  
  
    >>> fib(2)  
    1  
    >>> fib(10)  
    55  
    >>> fib(0)  
    0  
    >>> fib(3)  
    2  
    >>> fib(4)  
    3  
    >>> fib(8)  
    21  
    >>> fib(5)
```

Print Debugging

Print Debugging: Why do it?

- Simple and easy!
- Quickly gives you an insight into what is going on
- Does not require you to have an invariant in mind

```
def fact(x):  
    assert isinstance(x, int)  
    assert x >= 0  
    print("x =", x)  
    if x == 0:  
        return 1  
    else:  
        return x * fact(x - 1)
```

```
def half_fact(x):  
    return fact(x / 2)
```

Print Debugging: ok integration

- The code on the right doesn't work, if you have an ok test for fact(2)

Error: expected
2

but got

x= 2

x= 1

x= 0

2

```
def fact(x):  
    print("Debug: x=", x)  
    if x == 0:  
        return 1  
    else:  
        return x * fact(x - 1)
```

```
def half_fact(x):  
    return fact(x / 2)
```

Interactive Debugging

Interactive Debugging

- Sometimes you don't want to run the code every time you change what you choose to print
- Interactive debugging is live

Interactive Debugging: REPL

- The interactive mode of python, known as the REPL, is a useful tool
- To use, run
 - `python3 -i file.py`
 - then run whatever python commands you want
- OK integration:
 - `python3 ok -q whatever -i`
 - Starts out already having run code for that question

```
$ python3 -i lab00.py
>>> twenty_twenty()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "lab00.py", line 8, in twenty_twenty
        return _____
NameError: name '_____' is not defined
>>> 2020
2020
>>>
$ python3 ok -q twenty_twenty -i
=====
Assignment: Lab 0
OK, version v1.15.0
=====

-----Running tests-----

Doctests for twenty_twenty

>>> from lab00 import *
>>> twenty_twenty()
Traceback (most recent call last):
  File "/home/kavi/Downloads/lab00/lab00.py", line 8, in twenty_twenty
    return _____
NameError: name '_____' is not defined

# Error: expected
#     2020
# but got
#     Traceback (most recent call last):
#     ...
#     NameError: name '_____' is not defined

# Interactive console. Type exit() to quit
>>> 2020
2020
>>>
now exiting InteractiveConsole...
-----
Test summary
  0 test cases passed before encountering first failed test case

Backup... 100% complete
782068.283303
782068
Backup past deadline by 9 days, 1 hour, 14 minutes, and 28 seconds
Backup successful for user: kavi@berkeley.edu

OK is up to date
$
```

Interactive Debugging: PythonTutor

- You can also step through your code line by line on PythonTutor
 - Just copy your code into tutor.cs61a.org
- Ok integration
 - python ok -q whatever --trace

The screenshot shows the PythonTutor interface. On the left, a code editor displays `lab00.py` with the following content:

```
1 def twenty_twenty():
2     """Come up with the most creative expression that evaluates to 2020,
3     using only numbers and the +, *, and - operators.
4
5     >>> twenty_twenty()
6     2020
7     """
8     return _____
```

A yellow bar highlights the line `return _____`. Below the code editor, a status bar says "line that has just executed next line to execute". A red note at the bottom says "NEW! Click on a line of code to set a breakpoint. Then use the Forward and Back buttons to jump there."

On the right, the PythonTutor interface shows the execution state:

Frames	Objects
Global frame	function twenty_twenty() [parent=Global]
f1: twenty_twenty [parent=Global]	

An arrow points from the `twenty_twenty` entry in the Global frame to the corresponding function definition in the Objects section.

Error Types

Error Message Patterns

- Ideally: this wouldn't be necessary
 - Error messages would clearly say what they mean
- In practice, error messages are messy
- Not universal laws of nature (or even Python)
 - Good guidelines that are true >90% of the time

SyntaxError

- What it technically means
 - The file you ran isn't valid python syntax
- What it practically means
 - You made a typo
- What you should look for
 - Extra or missing parentheses
 - Missing colon at the end of an if or while statement
 - You started writing a statement but forgot to put anything inside

IndentationError

- What it technically means
 - The file you ran isn't valid python syntax, because of indentation inconsistency
- What it practically means
 - You used the wrong text editor
- What you should look for
 - You made a typo and misaligned something
 - You accidentally mixed tabs and spaces
 - Open your file in an editor that shows them
 - You used the wrong kind of spaces
 - Yes, there is more than one kind of space
 - If you think this is what's going on post on piazza with a link to the okpy backup

TypeError: ... 'X' object is not callable ...

- What it technically means
 - Objects of type X cannot be treated as functions
- What it practically means
 - You accidentally called a non-function as if it were a function
- What you should look for
 - Variables that should be functions being assigned to non-functions
 - Local variables that do not contain functions having the same name as functions in the global frame

TypeError: ... NoneType ...

- What it technically means
 - You used None in some operation it wasn't meant for
- What it practically means
 - You forgot a return statement in a function
- What you should look for
 - Functions missing return statements

NameError or UnboundLocalError

- What it technically means
 - Python looked up a name but didn't find it
- What it practically means
 - You made a typo
- What you should look for
 - A typo in the name in the description
 - (*less common*) Maybe you shadowed a variable from the global frame in a local frame (see right)

```
def f(x):  
    return x ** 2  
  
def g(x):  
    y = f(x)  
    def f():  
        return y + x  
    return f
```

Tracebacks

Parts of a Traceback

- Components
 - The error message itself
 - Lines #s on the way to the error
 - What's on those lines
- Most recent call is at the bottom

```
def f(x):  
    1 / 0  
def g(x):  
    f(x)  
def h(x):  
    g(x)  
print(h(2))
```

```
Traceback (most recent call last):  
  File "temp.py", line 7, in <module>  
    print(h(2))  
  File "temp.py", line 6, in h  
    g(x)  
  File "temp.py", line 4, in g  
    f(x)  
  File "temp.py", line 2, in f  
    1 / 0  
ZeroDivisionError: division by zero
```

How to read a traceback

```
def f(x):  
    1 / 0  
def g(x):  
    f(x)  
def h(x):  
    g(x)  
print(h(2))
```

1. Read the **error message**
 - a. Remember what common error messages mean!
2. Look at **each line**, bottom to top and see which one might be causing it

```
Traceback (most recent call last):  
  File "temp.py", line 7, in <module>  
    print(h(2))  
  File "temp.py", line 6, in h  
    g(x)  
  File "temp.py", line 4, in g  
    f(x)  
  File "temp.py", line 2, in f  
    1 / 0  
ZeroDivisionError: division by zero
```

Box-and-Pointer Notation

The Closure Property of Data Types

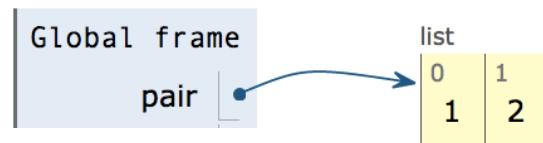
- A method for combining data values satisfies the *closure property* if:
The result of combination can itself be combined using the same method
- Closure is powerful because it permits us to create hierarchical structures
- Hierarchical structures are made up of parts, which themselves are made up of parts, and so on

Lists can contain lists as elements (in addition to anything else)

Box-and-Pointer Notation in Environment Diagrams

Lists are represented as a row of index-labeled adjacent boxes, one per element

Each box either contains a primitive value or points to a compound value



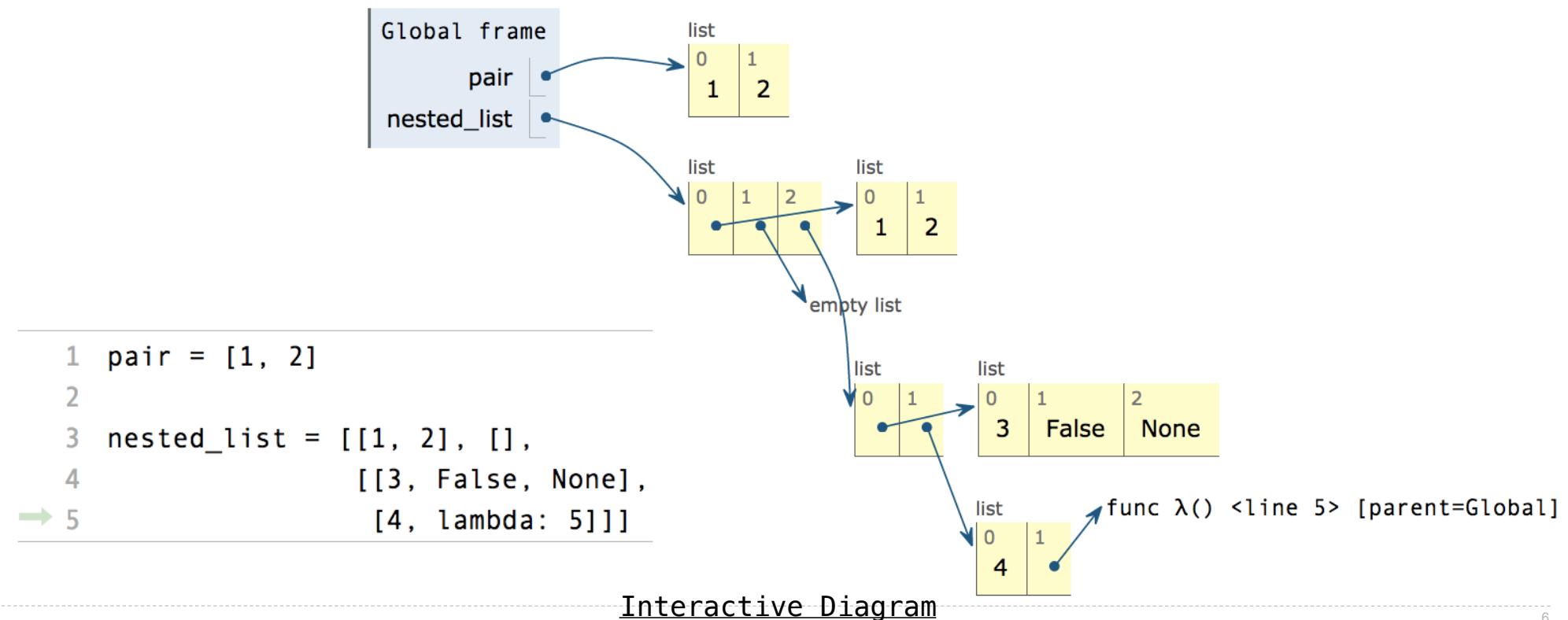
```
pair = [1, 2]
```

[Interactive Diagram](#)

Box-and-Pointer Notation in Environment Diagrams

Lists are represented as a row of index-labeled adjacent boxes, one per element

Each box either contains a primitive value or points to a compound value

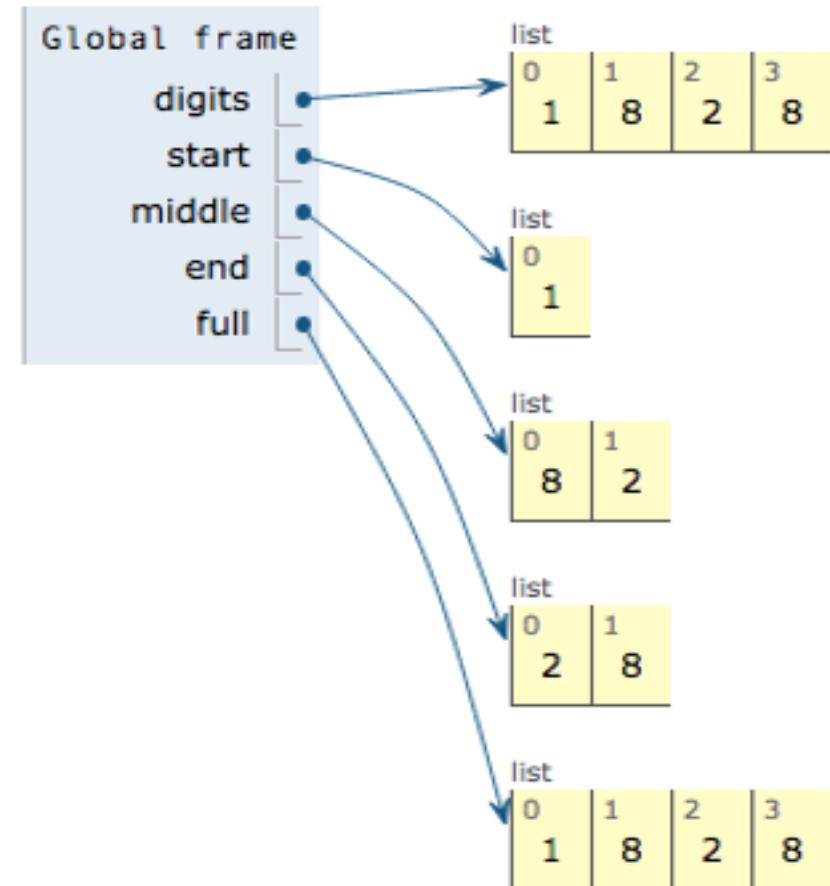


Slicing

(Demo)

Slicing Creates New Values

```
1 digits = [1, 8, 2, 8]
2 start = digits[:1]
3 middle = digits[1:3]
4 end = digits[2:]
→ 5 full = digits[:]
```



[Interactive Diagram](#)

Processing Container Values

Sequence Aggregation

Several built-in functions take iterable arguments and aggregate them into a value

- **sum(iterable[, start])** → value

Return the sum of an iterable of numbers (NOT strings) plus the value of parameter 'start' (which defaults to 0). When the iterable is empty, return start.

- **max(iterable[, key=func])** → value
max(a, b, c, ...[, key=func]) → value

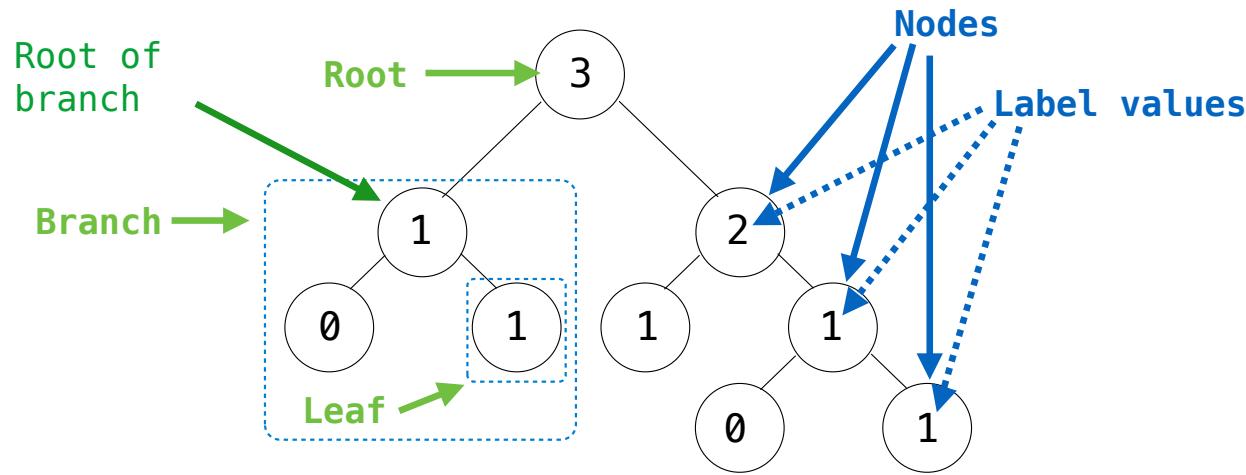
With a single iterable argument, return its largest item.
With two or more arguments, return the largest argument.

- **all(iterable)** → bool

Return True if `bool(x)` is True for all values x in the iterable.
If the iterable is empty, return True.

Trees

Tree Abstraction



Recursive description (wooden trees):

A **tree** has a **root** and a list of **branches**

Each branch is a **tree**

A tree with zero branches is called a **leaf**

Relative description (family trees):

Each location in a tree is called a **node**

Each **node** has a **label value**

One node can be the **parent/child** of another

People often refer to values by their locations: "each parent is the sum of its children"

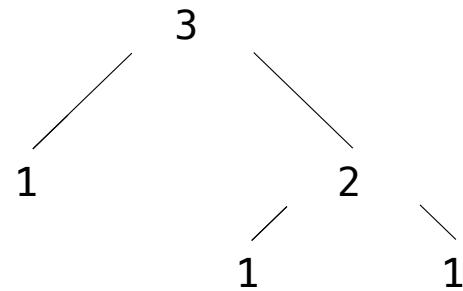
Implementing the Tree Abstraction

```
def tree(label, branches=[]):
    return [label] + branches
```

```
def label(tree):
    return tree[0]
```

```
def branches(tree):
    return tree[1:]
```

- A tree has a label value and a list of branches



```
>>> tree(3, [tree(1),
...             tree(2, [tree(1),
...                         tree(1)]))]
[3, [1], [2, [1], [1]]]
```

Implementing the Tree Abstraction

```
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)
```

Creates a list from a sequence of branches

```
def label(tree):
    return tree[0]
```

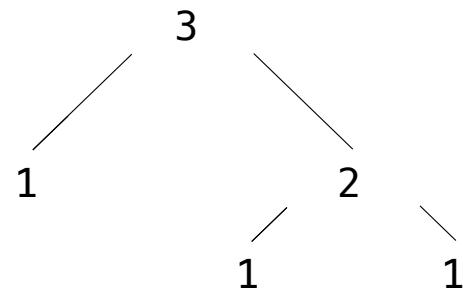
Verifies that tree is bound to a list

```
def branches(tree):
    return tree[1:]
```

Verifies the tree definition

```
def is_tree(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for branch in branches(tree):
        if not is_tree(branch):
            return False
    return True
```

- A tree has a label value and a list of branches



```
>>> tree(3, [tree(1),
...             tree(2, [tree(1),
...                         tree(1)]))]
[3, [1], [2, [1], [1]]]
```

```
def is_leaf(tree):
    return not branches(tree)
```

(Demo)

Tree Processing

(Demo)

Tree Processing Uses Recursion

Processing a leaf is often the base case of a tree processing function

The recursive case typically makes a recursive call on each branch, then aggregates

```
def count_leaves(t):
    """Count the leaves of a tree."""
    if is_leaf(t):
        return 1
    else:
        branch_counts = [count_leaves(b) for b in branches(t)]
        return sum(branch_counts)
```

(Demo)

Discussion Question

Implement `leaves`, which returns a list of the leaf labels of a tree

Hint: If you `sum` a list of lists, you get a list containing the elements of those lists

```
>>> sum([ [1], [2, 3], [4] ], [])
[1, 2, 3, 4]
>>> sum([ [1] ], [])
[1]
>>> sum([ [[1]], [2] ], [])
[[1], 2]
```

```
def leaves(tree):
    """Return a list containing the leaves of tree.

    >>> leaves(fib_tree(5))
    [1, 0, 1, 0, 1, 1, 0, 1]
    """
    if is_leaf(tree):
        return [label(tree)]
    else:
        return sum(List of leaves for each branch, [])
```

branches(tree)	[b for b in branches(tree)]
leaves(tree)	[s for s in leaves(tree)]
[branches(b) for b in branches(tree)]	[branches(s) for s in leaves(tree)]
[leaves(b) for b in branches(tree)]	[leaves(s) for s in leaves(tree)]

Creating Trees

A function that creates a tree from another tree is typically also recursive

```
def increment_leaves(t):
    """Return a tree like t but with leaf values incremented."""
    if is_leaf(t):
        return tree(label(t) + 1)
    else:
        bs = [increment_leaves(b) for b in branches(t)]
        return tree(label(t), bs)

def increment(t):
    """Return a tree like t but with all node values incremented."""
    return tree(label(t) + 1, [increment(b) for b in branches(t)])
```

Example: Printing Trees

(Demo)

Objects

(Demo)

Objects

- Objects represent information
- They consist of data and behavior, bundled together to create abstractions
- Objects can represent things, but also properties, interactions, & processes
- A type of object is called a class; **classes** are first-class values in Python
- Object-oriented programming:
 - A metaphor for organizing large programs
 - Special syntax that can improve the composition of programs
- In Python, every value is an object
 - All **objects** have **attributes**
 - A lot of data manipulation happens through object **methods**
 - Functions do one thing; objects do many related things

Example: Strings

(Demo)

Representing Strings: the ASCII Standard

American Standard Code for Information Interchange

ASCII Code Chart															
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
!	"	#	\$	%	&	'	()	*	+	,	-	.	/	
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	-
~	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
p	q	r	s	t	u	v	w	x	y	z	{		}	-	DEL

8 rows: 3 bits

16 columns: 4 bits

"Bell" (\a)

"Line feed" (\n)

- Layout was chosen to support sorting by character code
- Rows indexed 2–5 are a useful 6-bit (64 element) subset
- Control characters were designed for transmission

(Demo)

Representing Strings: the Unicode Standard

- 109,000 characters
- 93 scripts (organized)
- Enumeration of character properties, such as case
- Supports bidirectional display order
- A canonical name for every character

聳	聲	聳	聽	蹠	聶	職	瞻
8071	8072	8073	8074	8075	8076	8077	8078
健	膾	腳	腴	暇	脰	脢	腸
8171	8172	8173	8174	8175	8176	8177	8178
艱	色	艳	艷	艷	艷	艷	艷
8271	8272	8273	8274	8275	8276	8277	8278
毫	莖	荳	莢	葱	苓	荷	荸
8371	8372	8373	8374	8375	8376	8377	8378
葱	莧	葳	歲	葵	葶	葷	蕙

http://ian-albert.com/unicode_chart/unichart-chinese.jpg

U+0058 LATIN CAPITAL LETTER X

U+263a WHITE SMILING FACE

U+2639 WHITE FROWNING FACE



(Demo)

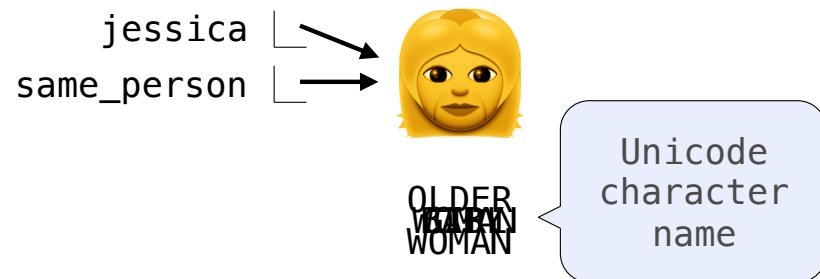
Mutation Operations

Some Objects Can Change

[Demo]

First example in the course of an object changing state

The same object can change in value throughout the course of computation



All names that refer to the same object are affected by a mutation

Only objects of *mutable* types can change: lists & dictionaries

{Demo}

Mutation Can Happen Within a Function Call

A function can change the value of any object in its scope

```
>>> four = [1, 2, 3, 4]
>>> len(four)
4
>>> mystery(four)
>>> len(four)
2
```

```
def mystery(s): or def mystery(s):
    s.pop()           s[2:] = []
    s.pop()
```

```
>>> four = [1, 2, 3, 4]
>>> len(four)
4
>>> another_mystery() # No arguments!
>>> len(four)
2
```

```
def another_mystery():
    four.pop()
    four.pop()
```

[Interactive Diagram](#)

Tuples

(Demo)

Tuples are Immutable Sequences

Immutable values are protected from mutation

```
>>> turtle = (1, 2, 3)
>>> ooze()
>>> turtle
(1, 2, 3)
```

Next lecture: ooze can change turtle's binding

```
>>> turtle = [1, 2, 3]
>>> ooze()
>>> turtle
['Anything could be inside!']
```

The value of an expression can change because of changes in names or objects

Name change:

```
>>> x = 2
>>> x + x
4
>>> x = 3
>>> x + x
6
```

Object mutation:

```
>>> x = [1, 2]
>>> x + x
[1, 2, 1, 2]
>>> x.append(3)
>>> x + x
[1, 2, 3, 1, 2, 3]
```

An immutable sequence may still change if it *contains* a mutable value as an element

```
>>> s = ([1, 2], 3)
>>> s[0] = 4
ERROR
```

```
>>> s = ([1, 2], 3)
>>> s[0][0] = 4
>>> s
([4, 2], 3)
```

Mutation

Sameness and Change

- As long as we never modify objects, a compound object is just the totality of its pieces
- A rational number is just its numerator and denominator
- This view is no longer valid in the presence of change
- A compound data object has an "identity" in addition to the pieces of which it is composed
- A list is still "the same" list even if we change its contents
- Conversely, we could have two lists that happen to have the same contents, but are different

```
>>> a = [10]
>>> b = a
>>> a == b
True
>>> a.append(20)
>>> a == b
True
>>> a
[10, 20]
>>> b
[10, 20]
```

```
>>> a = [10]
>>> b = [10]
>>> a == b
True
>>> b.append(20)
>>> a
[10]
>>> b
[10, 20]
>>> a == b
False
```

Identity Operators

Identity

`<exp0> is <exp1>`

evaluates to `True` if both `<exp0>` and `<exp1>` evaluate to the same object

Equality

`<exp0> == <exp1>`

evaluates to `True` if both `<exp0>` and `<exp1>` evaluate to equal values

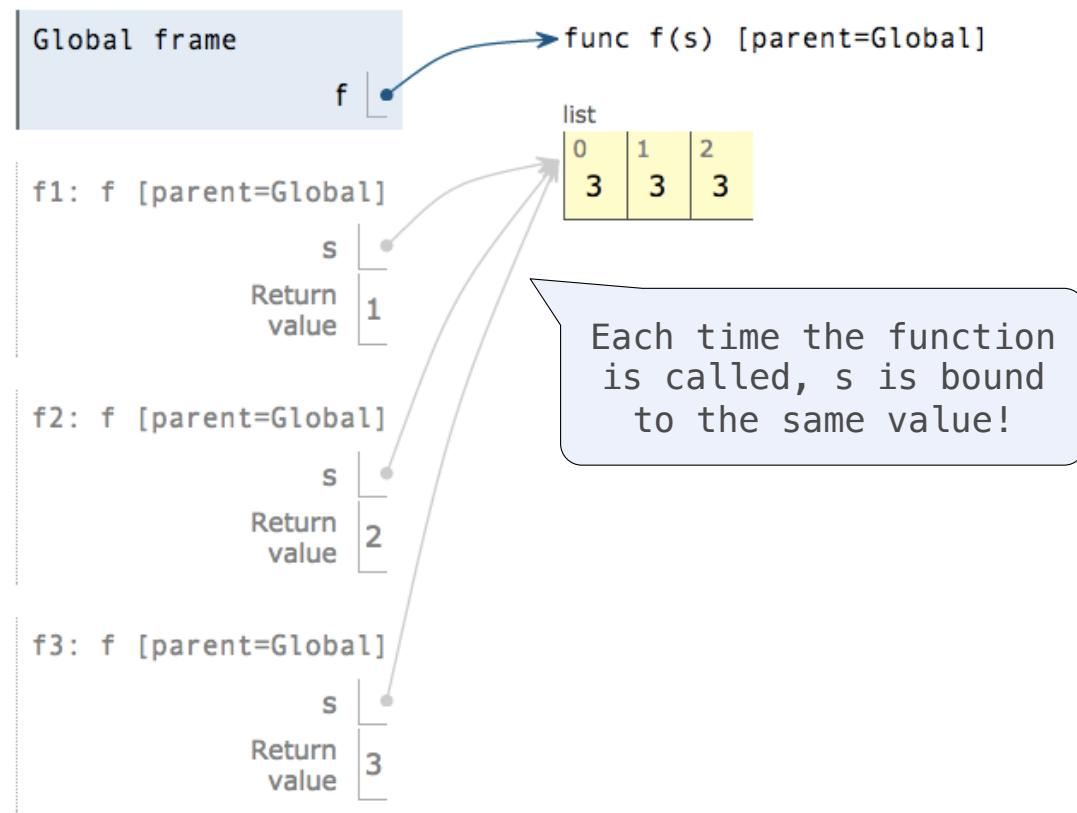
Identical objects are always equal values

(Demo)

Mutable Default Arguments are Dangerous

A default argument value is part of a function value, not generated by a call

```
>>> def f(s=[]):  
...     s.append(3)  
...     return len(s)  
  
>>> f()  
1  
>>> f()  
2  
>>> f()  
3
```



[Interactive Diagram](#)

Mutable Functions

A Function with Behavior That Varies Over Time

Let's model a bank account that has a balance of \$100

Return value:
remaining balance

```
>>> withdraw(25)  
75
```

Argument:
amount to withdraw

Different
return value!

```
>>> withdraw(25)  
50  
>>> withdraw(60)  
'Insufficient funds'
```

Second withdrawal of
the same amount

```
>>> withdraw(15)  
35
```

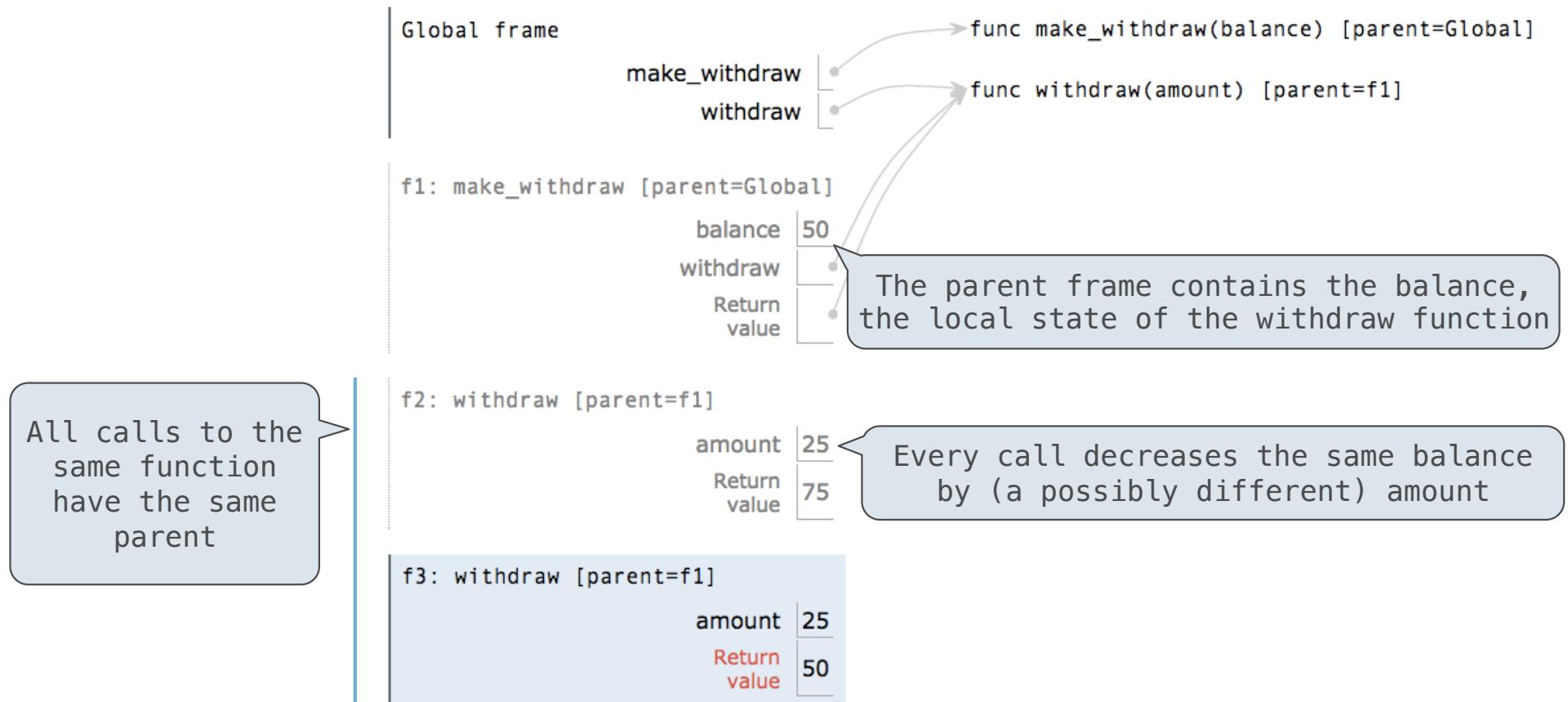
Where's this balance
stored?

```
>>> withdraw = make_withdraw(100)
```

Within the parent frame
of the function!

A function has a body and
a parent environment

Persistent Local State Using Environments

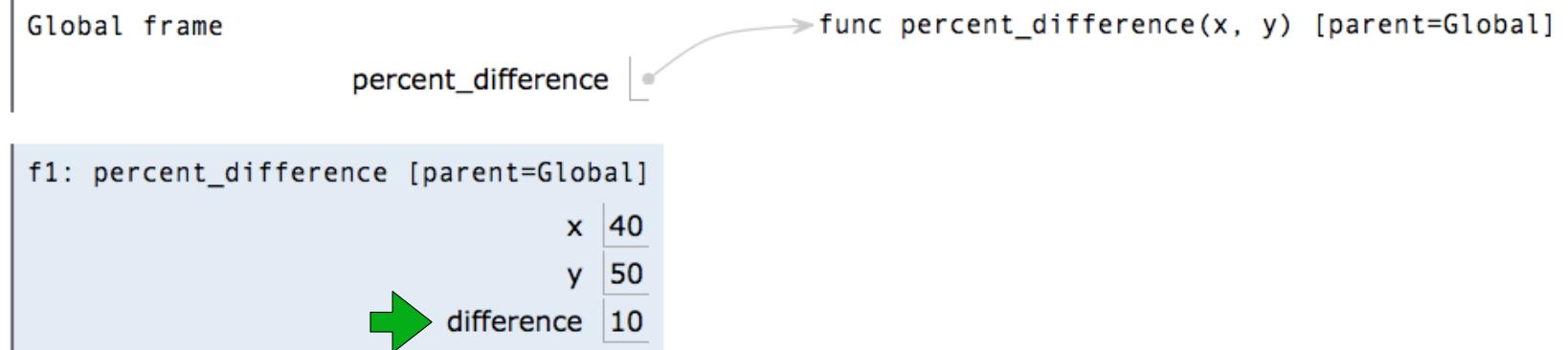


[pythontutor.com/composingprograms.html#code=def%20make_withdraw%28balance%29%3A%0A%20%20def%20withdraw%28amount%29%3A%0A%20%20%20%20%20nonlocal%20balance%0A%20%20%20%20%20if%20amount%28-%20balance%3A%0A%20%20%20%20%20return%20'Insufficient%20funds'%0A%20%20%20%20%20return%20balance%0Awithdraw%20%3D%20make_withdraw%28100%29%0Awithdraw%2825%29%0Awithdraw%2825%29%0Awithdraw%2815%29%0Awithdraw%2815%29&mode=display&origin=composingprograms.js&cumulative=true&py=3&rawInputListJSOn=%5B%5D&curInstr=0](http://pythontutor.com/composingprograms.html#code=def%20make_withdraw%28balance%29%3A%0A%20%20def%20withdraw%28amount%29%3A%0A%20%20%20%20%20nonlocal%20balance%0A%20%20%20%20%20if%20amount%28-%20balance%3A%0A%20%20%20%20%20return%20'Insufficient%20funds'%0A%20%20%20%20%20return%20balance%0A%20%20%20%20%20return%20withdraw%0Awithdraw%20%3D%20make_withdraw%28100%29%0Awithdraw%2825%29%0Awithdraw%2825%29%0Awithdraw%2815%29%0Awithdraw%2815%29&mode=display&origin=composingprograms.js&cumulative=true&py=3&rawInputListJSOn=%5B%5D&curInstr=0)

Reminder: Local Assignment

```
def percent_difference(x, y):
    difference = abs(x-y)
    return 100 * difference / x
diff = percent_difference(40, 50)
```

Assignment binds name(s) to value(s) in the first frame of the current environment



Execution rule for assignment statements:

1. Evaluate all expressions right of `=`, from left to right
2. Bind the names on the left to the resulting values in the **current frame**

Non-Local Assignment & Persistent Local State

```
def make_withdraw(balance):  
    """Return a withdraw function with a starting balance."""  
  
    def withdraw(amount):  
        nonlocal balance  
        if amount > balance:  
            return 'Insufficient funds'  
        balance = balance - amount  
        return balance  
  
    return withdraw
```

nonlocal balance Declare the name "balance" nonlocal at the top of the body of the function in which it is re-assigned

balance = balance - amount Re-bind balance in the first non-local frame in which it was bound previously

(Demo)

Non-Local Assignment

The Effect of Nonlocal Statements

`nonlocal <name>, <name>, ...`

Effect: Future assignments to that name change its pre-existing binding in the **first non-local frame** of the current environment in which that name is bound.

Python Docs: an
"enclosing scope"

From the Python 3 language reference:

Names listed in a nonlocal statement must refer to pre-existing bindings in an enclosing scope.

Names listed in a nonlocal statement must not collide with pre-existing bindings in the **local scope**.

Current frame

http://docs.python.org/release/3.1.3/reference/simple_stmts.html#the-nonlocal-statement

<http://www.python.org/dev/peps/pep-3104/>

The Many Meanings of Assignment Statements

x = 2

Status

Effect

- No nonlocal statement
- "x" **is not** bound locally

Create a new binding from name "x" to object 2 in the first frame of the current environment

-
- No nonlocal statement
 - "x" **is** bound locally

Re-bind name "x" to object 2 in the first frame of the current environment

-
- nonlocal x
 - "x" **is** bound in a non-local frame

Re-bind "x" to 2 in the first non-local frame of the current environment in which "x" is bound

-
- nonlocal x
 - "x" **is not** bound in a non-local frame

SyntaxError: no binding for nonlocal 'x' found

-
- nonlocal x
 - "x" **is** bound in a non-local frame
 - "x" also bound locally

SyntaxError: name 'x' is parameter and nonlocal

Python Particulars

Python pre-computes which frame contains each name before executing the body of a function.

Within the body of a function, all instances of a name must refer to the same frame.

```
def make_withdraw(balance):
    def withdraw(amount):
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount
        return balance
    return withdraw
```

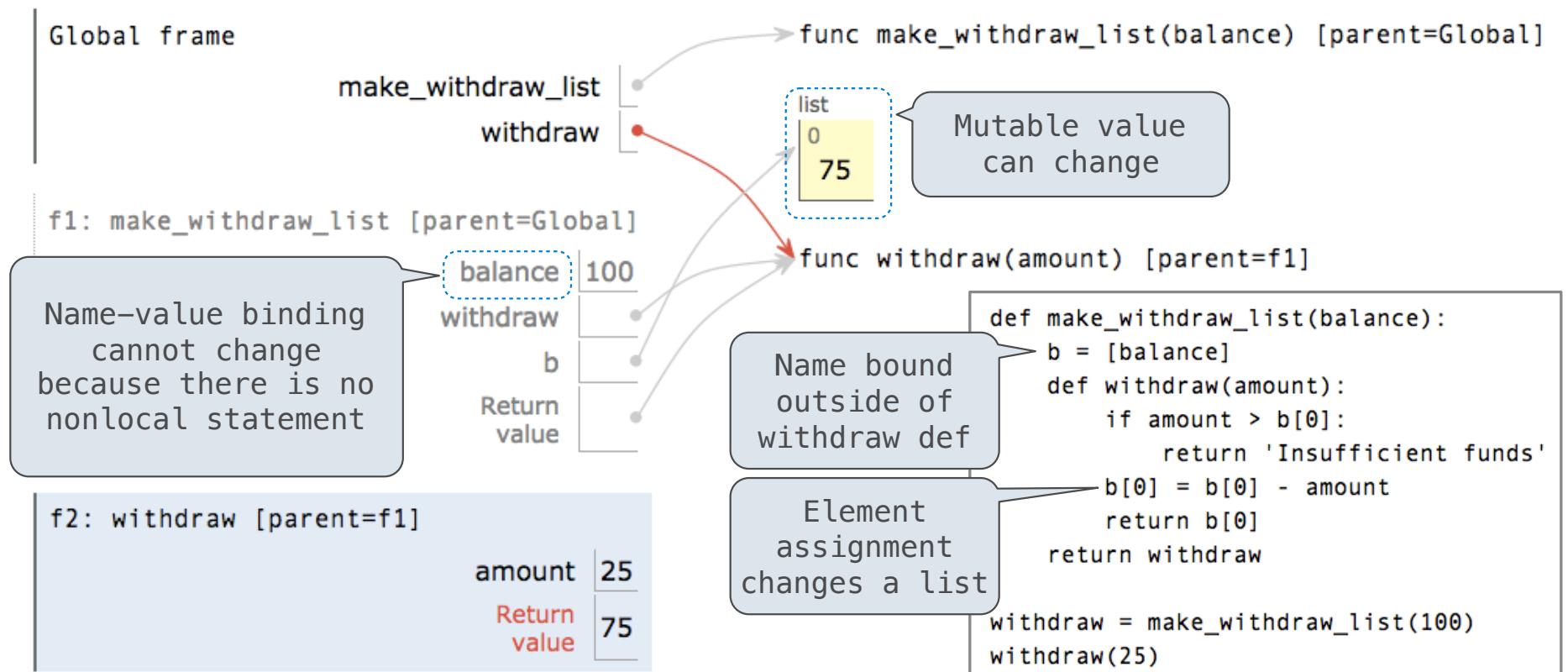
Local assignment

```
wd = make_withdraw(20)
wd(5)
```

`UnboundLocalError: local variable 'balance' referenced before assignment`

Mutable Values & Persistent Local State

Mutable values can be changed *without* a nonlocal statement.



goo.gl/y4TyFZ

12

Multiple Mutable Functions

(Demo)

Referential Transparency, Lost

- Expressions are **referentially transparent** if substituting an expression with its value does not change the meaning of a program.



```
mul(add(2, mul(4, 6)), add(3, 5))  
  
mul(add(2,      24      ), add(3, 5))  
  
mul(      26      , add(3, 5))
```



- Mutation operations violate the condition of referential transparency because they do more than just return a value; **they change the environment.**

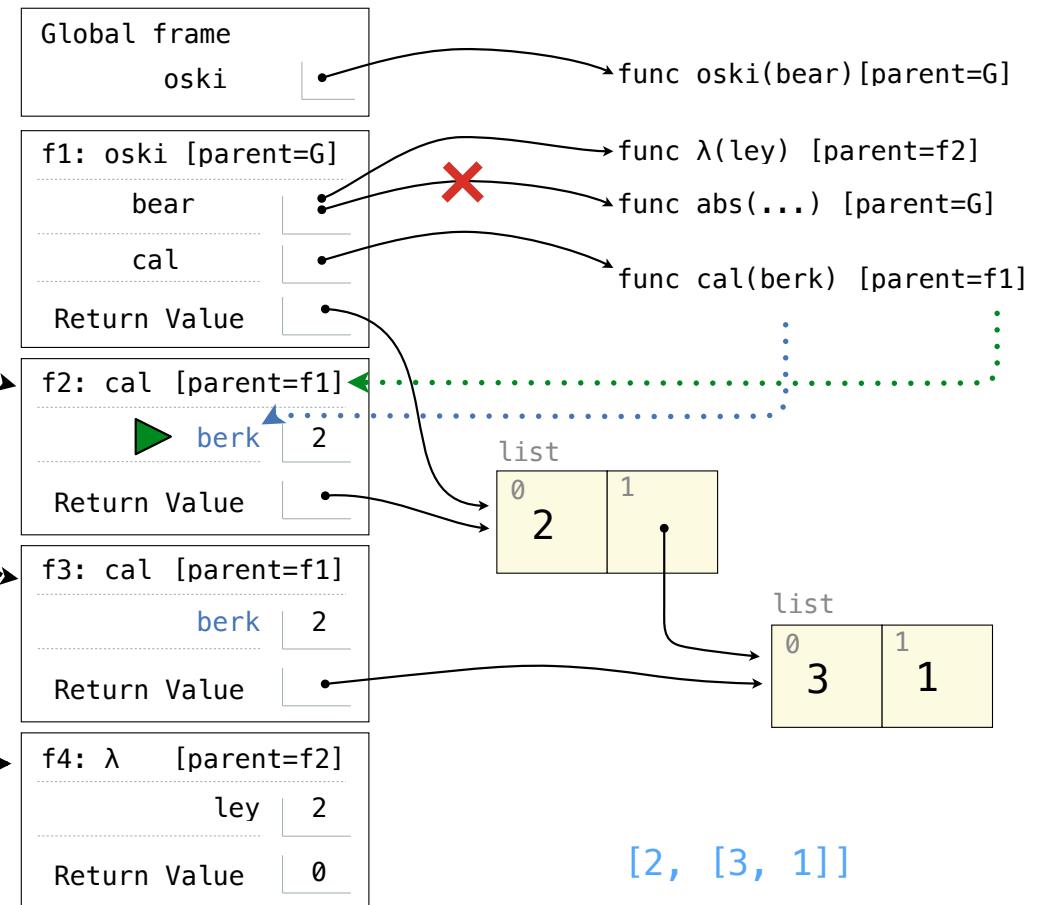
Environment Diagrams

Go Bears!

```

def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)
oski(abs)

```



Iterators

Iterators

A container can provide an iterator that provides access to its elements in order

`iter(iterable)`: Return an iterator over the elements
of an iterable value

`next(iterator)`: Return the next element in an iterator

```
>>> s = [3, 4, 5]           ▼  
>>> t = iter(s)  
>>> next(t)  
3  
>>> next(t)  
4  
>>> u = iter(s)  
>>> next(u)  
3  
>>> next(t)  
5  
>>> next(u)  
4
```

(Demo)

Dictionary Iteration

Views of a Dictionary

An *iterable* value is any value that can be passed to `iter` to produce an iterator

An *iterator* is returned from `iter` and can be passed to `next`; all iterators are mutable

A dictionary, its keys, its values, and its items are all iterable values

- The order of items in a dictionary is the order in which they were added (Python 3.6+)
- Historically, items appeared in an arbitrary order (Python 3.5 and earlier)

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys()) # or iter(d)
>>> next(k)
'one'
>>> next(k)
'two'
>>> next(k)


```
>>> v = iter(d.values())
>>> next(v)
1
>>> next(v)
2
>>> next(v)
3
>>> next(v)
0
```



```
>>> i = iter(d.items())
>>> next(i)
('one', 1)
>>> next(i)
('two', 2)
>>> next(i)
('three', 3)
>>> next(i)
('zero', 0)
```


```

(Demo)

For Statements

(Demo)

Built-In Iterator Functions

Built-in Functions for Iteration

Many built-in Python sequence operations return iterators that compute results lazily

<code>map(func, iterable):</code>	Iterate over <code>func(x)</code> for <code>x</code> in <code>iterable</code>
<code>filter(func, iterable):</code>	Iterate over <code>x</code> in <code>iterable</code> if <code>func(x)</code>
<code>zip(first_iter, second_iter):</code>	Iterate over co-indexed <code>(x, y)</code> pairs
<code>reversed(sequence):</code>	Iterate over <code>x</code> in a sequence in reverse order

To view the contents of an iterator, place the resulting elements into a container

<code>list(iterable):</code>	Create a list containing all <code>x</code> in <code>iterable</code>
<code>tuple(iterable):</code>	Create a tuple containing all <code>x</code> in <code>iterable</code>
<code>sorted(iterable):</code>	Create a sorted list containing <code>x</code> in <code>iterable</code> (Demo)

Generators

Generators and Generator Functions

```
>>> def plus_minus(x):
...     yield x
...     yield -x

>>> t = plus_minus(3)
>>> next(t)
3
>>> next(t)
-3
>>> t
<generator object plus_minus ...>
```

A *generator function* is a function that **yields** values instead of **returning** them

A normal function **returns** once; a *generator function* can **yield** multiple times

A *generator* is an iterator created automatically by calling a *generator function*

When a *generator function* is called, it returns a *generator* that iterates over its yields

(Demo)

Generators & Iterators

Generators can Yield from Iterators

A `yield from` statement yields all values from an iterator or iterable (Python 3.3)

```
>>> list(a_then_b([3, 4], [5, 6]))  
[3, 4, 5, 6]
```

```
def a_then_b(a, b):           def a_then_b(a, b):  
    for x in a:               yield from a  
        yield x              yield from b  
    for x in b:  
        yield x
```

```
>>> list(countdown(5))  
[5, 4, 3, 2, 1]
```

```
def countdown(k):  
    if k > 0:  
        yield k  
        yield from countdown(k-1)
```

(Demo)

Object-Oriented Programming

Object-Oriented Programming

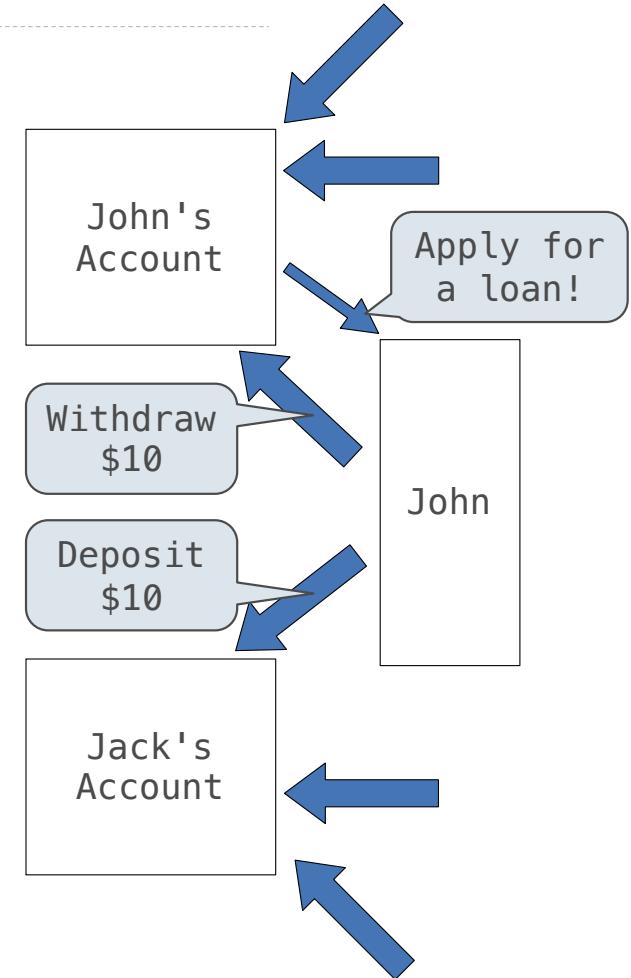
A method for organizing programs

- Data abstraction
- Bundling together information and related behavior

A metaphor for computation using distributed state

- Each object has its own local state
- Each object also knows how to manage its own local state, based on method calls
- Method calls are messages passed between objects
- Several objects may all be instances of a common type
- Different types may relate to each other

Specialized syntax & vocabulary to support this metaphor



Classes

A class serves as a template for its instances

Idea: All bank accounts have a `balance` and an account `holder`; the `Account` class should add those attributes to each newly created instance

```
>>> a = Account('John')
>>> a.holder
'John'
>>> a.balance
0
```

Idea: All bank accounts should have `withdraw` and `deposit` behaviors that all work in the same way

```
>>> a.deposit(15)
15
>>> a.withdraw(10)
5
>>> a.balance
5
>>> a.withdraw(10)
'Insufficient funds'
```

Better idea: All bank accounts share a `withdraw` method and a `deposit` method

Class Statements

The Class Statement

```
class <name>:  
    <suite>
```

The suite is executed when the class statement is executed.

A class statement creates a new class and binds that class to <name> in the first frame of the current environment

Assignment & def statements in <suite> create attributes of the class (not names in frames)

```
>>> class Clown:  
...     nose = 'big and red'  
...     def dance():  
...         return 'No thanks'  
...  
>>> Clown.nose  
'big and red'  
>>> Clown.dance()  
'No thanks'  
>>> Clown  
<class '__main__.Clown'>
```

Object Construction

Idea: All bank accounts have a **balance** and an account **holder**; the **Account** class should add those attributes to each of its instances

```
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```

When a class is called:

1. A new instance of that class is created: **An account instance**

balance: 0 holder: 'Jim'

2. The **__init__** method of the class is called with the new object as its first argument (named **self**), along with any additional arguments provided in the call expression

__init__ is called
a constructor

```
class Account:
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
```

Object Identity

Every object that is an instance of a user-defined class has a unique identity:

```
>>> a = Account('John')
>>> b = Account('Jack')
>>> a.balance
0
>>> b.holder
'Jack'
```

Every call to Account creates a new Account instance. There is only one Account class.

Identity operators "is" and "is not" test if two expressions evaluate to the same object:

```
>>> a is a
True
>>> a is not b
True
```

Binding an object to a new name using assignment does not create a new object:

```
>>> c = a
>>> c is a
True
```

Methods

Methods

Methods are functions defined in the suite of a class statement

```
class Account:  
    def __init__(self, account_holder):  
        self.balance = 0  
        self.holder = account_holder  
  
        self should always be bound to an instance of the Account class  
  
    def deposit(self, amount):  
        self.balance = self.balance + amount  
        return self.balance  
    def withdraw(self, amount):  
        if amount > self.balance:  
            return 'Insufficient funds'  
        self.balance = self.balance - amount  
        return self.balance
```

These def statements create function objects as always,
but their names are bound as attributes of the class

Invoking Methods

All invoked methods have access to the object via the `self` parameter, and so they can all access and manipulate the object's state

```
class Account:  
    ...  
    def deposit(self, amount):  
        self.balance = self.balance + amount  
        return self.balance
```

Defined with two parameters

Dot notation automatically supplies the first argument to a method

```
>>> tom_account = Account('Tom')  
>>> tom_account.deposit(100)  
100
```

Bound to self

Invoked with one argument

Dot Expressions

Objects receive messages via dot notation

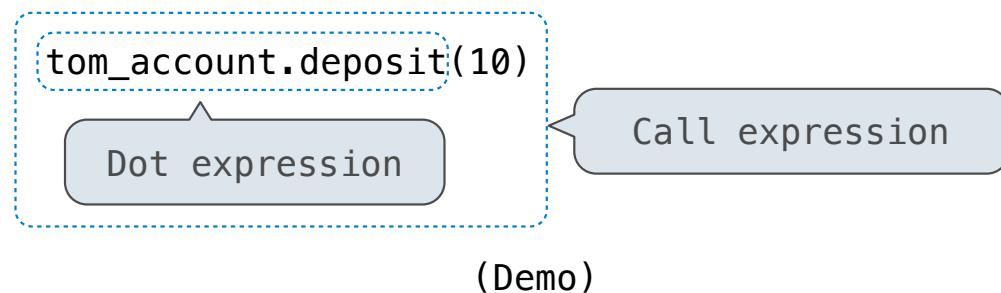
Dot notation accesses attributes of the instance or its class

`<expression> . <name>`

The `<expression>` can be any valid Python expression

The `<name>` must be a simple name

Evaluates to the value of the attribute looked up by `<name>` in the object that is the value of the `<expression>`



Attributes

(Demo)

Accessing Attributes

Using `getattr`, we can look up an attribute using a string

```
>>> getattr(tom_account, 'balance')
10

>>> hasattr(tom_account, 'deposit')
True
```

`getattr` and dot expressions look up a name in the same way

Looking up an attribute name in an object may return:

- One of its instance attributes, or
- One of the attributes of its class

Methods and Functions

Python distinguishes between:

- *Functions*, which we have been creating since the beginning of the course, and
- *Bound methods*, which couple together a function and the object on which that method will be invoked

Object + Function = Bound Method

```
>>> type(Account.deposit)
<class 'function'>
>>> type(tom_account.deposit)
<class 'method'>
```

```
>>> Account.deposit(tom_account, 1001)
1011
>>> tom_account.deposit(1004)
2015
```

Function: all arguments within parentheses

Method: One object before the dot and other arguments within parentheses

Looking Up Attributes by Name

<expression> . <name>

To evaluate a dot expression:

1. Evaluate the <expression> to the left of the dot, which yields the object of the dot expression
2. <name> is matched against the instance attributes of that object; if an attribute with that name exists, its value is returned
3. If not, <name> is looked up in the class, which yields a class attribute value
4. That value is returned unless it is a function, in which case a bound method is returned instead

Class Attributes

Class attributes are "shared" across all instances of a class because they are attributes of the class, not the instance

```
class Account:  
  
    interest = 0.02    # A class attribute  
  
    def __init__(self, account_holder):  
        self.balance = 0  
        self.holder = account_holder  
  
    # Additional methods would be defined here  
  
    >>> tom_account = Account('Tom')  
    >>> jim_account = Account('Jim')  
    >>> tom_account.interest  
0.02  
    >>> jim_account.interest  
0.02
```

The **interest** attribute is **not** part of the instance; it's part of the class!

Attributes

Methods and Functions

Python distinguishes between:

- *Functions*, which we have been creating since the beginning of the course, and
- *Bound methods*, which couple together a function and the object on which that method will be invoked

Object + Function = Bound Method

```
>>> type(Account.deposit)
<class 'function'>
>>> type(tom_account.deposit)
<class 'method'>
```

```
>>> Account.deposit(tom_account, 1001)
1011
>>> tom_account.deposit(1004)
2015
```

Function: all arguments within parentheses

Method: One object before the dot and other arguments within parentheses

Terminology: Attributes, Functions, and Methods

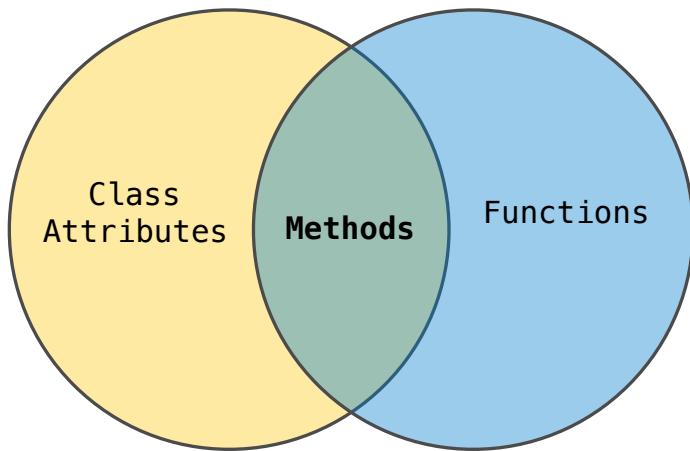
All objects have attributes, which are name-value pairs

Classes are objects too, so they have attributes

Instance attribute: attribute of an instance

Class attribute: attribute of the class of an instance

Terminology:



Python object system:

Functions are objects

Bound methods are also objects: a function that has its first parameter "self" already bound to an instance

Dot expressions evaluate to bound methods for class attributes that are functions

`<instance>.<method_name>`

Looking Up Attributes by Name

<expression> . <name>

To evaluate a dot expression:

1. Evaluate the <expression> to the left of the dot, which yields the object of the dot expression
2. <name> is matched against the instance attributes of that object; if an attribute with that name exists, its value is returned
3. If not, <name> is looked up in the class, which yields a class attribute value
4. That value is returned unless it is a function, in which case a bound method is returned instead

Class Attributes

Class attributes are "shared" across all instances of a class because they are attributes of the class, not the instance

```
class Account:  
  
    interest = 0.02    # A class attribute  
  
    def __init__(self, account_holder):  
        self.balance = 0  
        self.holder = account_holder  
  
    # Additional methods would be defined here  
  
    >>> tom_account = Account('Tom')  
    >>> jim_account = Account('Jim')  
    >>> tom_account.interest  
0.02  
    >>> jim_account.interest  
0.02
```

The **interest** attribute is **not** part of the instance; it's part of the class!

Attribute Assignment

Assignment to Attributes

Assignment statements with a dot expression on their left-hand side affect attributes for the object of that dot expression

- If the object is an instance, then assignment sets an instance attribute
- If the object is a class, then assignment sets a class attribute

```
class Account:  
    interest = 0.02  
    def __init__(self, holder):  
        self.holder = holder  
        self.balance = 0  
    ...  
  
tom_account = Account('Tom')
```

Instance
Attribute
Assignment

tom_account.interest = 0.08

This expression
evaluates to an
object

But the name ("interest")
is not looked up

Attribute
assignment
statement adds
or modifies the
attribute named
"interest" of
tom_account

Class
Attribute : Account.interest = 0.04
Assignment

Attribute Assignment Statements

Account class
attributes

interest: ~~0.02~~ ~~0.04~~ 0.05
(withdraw, deposit, __init__)

Instance
attributes of
jim_account

balance: 0
holder: 'Jim'
interest: 0.08

Instance
attributes of
tom_account

balance: 0
holder: 'Tom'

```
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
>>> tom_account.interest
0.02
>>> jim_account.interest
0.02
>>> Account.interest = 0.04
>>> tom_account.interest
0.04
>>> jim_account.interest
0.04
```

```
>>> jim_account.interest = 0.08
>>> jim_account.interest
0.08
>>> tom_account.interest
0.04
>>> Account.interest = 0.05
>>> tom_account.interest
0.05
>>> jim_account.interest
0.08
```

Inheritance

Inheritance

Inheritance is a technique for relating classes together

A common use: Two similar classes differ in their degree of specialization

The specialized class may have the same attributes as the general class, along with some special-case behavior

```
class <Name>(<Base Class>):  
    <suite>
```

Conceptually, the new subclass inherits attributes of its base class

The subclass may override certain inherited attributes

Using inheritance, we implement a subclass by specifying its differences from the the base class

Inheritance Example

A `CheckingAccount` is a specialized type of `Account`

```
>>> ch = CheckingAccount('Tom')
>>> ch.interest      # Lower interest rate for checking accounts
0.01
>>> ch.deposit(20)  # Deposits are the same
20
>>> ch.withdraw(5)  # Withdrawals incur a $1 fee
14
```

Most behavior is shared with the base class `Account`

```
class CheckingAccount(Account):
    """A bank account that charges for withdrawals."""
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_fee)
        ↑
        return super().withdraw(amount + self.withdraw_fee)
```

or

Looking Up Attribute Names on Classes

Base class attributes aren't copied into subclasses!

To look up a name in a class:

1. If it names an attribute in the class, return the attribute value.
2. Otherwise, look up the name in the base class, if there is one.

```
>>> ch = CheckingAccount('Tom') # Calls Account.__init__
>>> ch.interest      # Found in CheckingAccount
0.01
>>> ch.deposit(20)   # Found in Account
20
>>> ch.withdraw(5)   # Found in CheckingAccount
14
```

(Demo)

Object-Oriented Design

Designing for Inheritance

Don't repeat yourself; use existing implementations

Attributes that have been overridden are still accessible via class objects

Look up attributes on instances whenever possible

```
class CheckingAccount(Account):
    """A bank account that charges for withdrawals."""
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_fee)
```

Attribute look-up on base class

Preferred to CheckingAccount.withdraw_fee to allow for specialized accounts

Inheritance and Composition

Object-oriented programming shines when we adopt the metaphor

Inheritance is best for representing is-a relationships

- E.g., a checking account is a specific type of account
- So, CheckingAccount inherits from Account

Composition is best for representing has-a relationships

- E.g., a bank has a collection of bank accounts it manages
- So, A bank has a list of accounts as an attribute

(Demo)

Attributes Lookup Practice

Inheritance and Attribute Lookup

```
class A:  
    z = -1  
    def f(self, x):  
        return B(x-1)
```

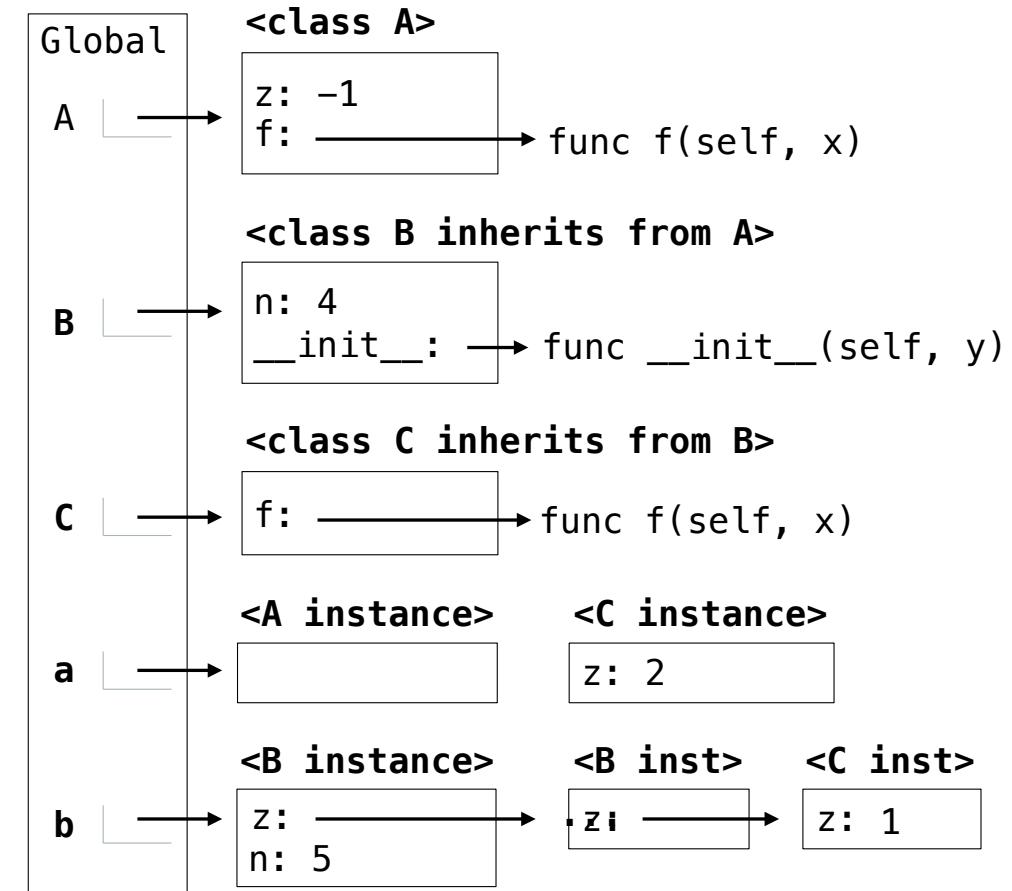
```
class B(A):  
    n = 4  
    def __init__(self, y):  
        if y:  
            self.z = self.f(y)  
        else:  
            self.z = C(y+1)
```

```
class C(B):  
    def f(self, x):  
        return x
```

```
a = A()  
b = B(1)  
b.n = 5
```

```
>>> C(2).n  
4  
>>> a.z == C.z  
True  
>>> a.z == b.z  
False
```

Which evaluates to an integer?
b.z
b.z.z
b.z.z.z
None of these



Multiple Inheritance

Multiple Inheritance

```
class SavingsAccount(Account):
    deposit_fee = 2
    def deposit(self, amount):
        return Account.deposit(self, amount - self.deposit_fee)
```

A class may inherit from multiple base classes in Python

CleverBank marketing executive has an idea:

- Low interest rate of 1%
- A \$1 fee for withdrawals
- A \$2 fee for deposits
- A free dollar when you open your account

```
class AsSeenOnTVAccount(CheckingAccount, SavingsAccount):
    def __init__(self, account_holder):
        self.holder = account_holder
        self.balance = 1                         # A free dollar!
```

Multiple Inheritance

A class may inherit from multiple base classes in Python.

```
class AsSeenOnTVAccount(CheckingAccount, SavingsAccount):
    def __init__(self, account_holder):
        self.holder = account_holder
        self.balance = 1                         # A free dollar!
```

Instance attribute

```
>>> such_a_deal = AsSeenOnTVAccount('John')
>>> such_a_deal.balance
```

1

SavingsAccount method

```
>>> such_a_deal.deposit(20)
```

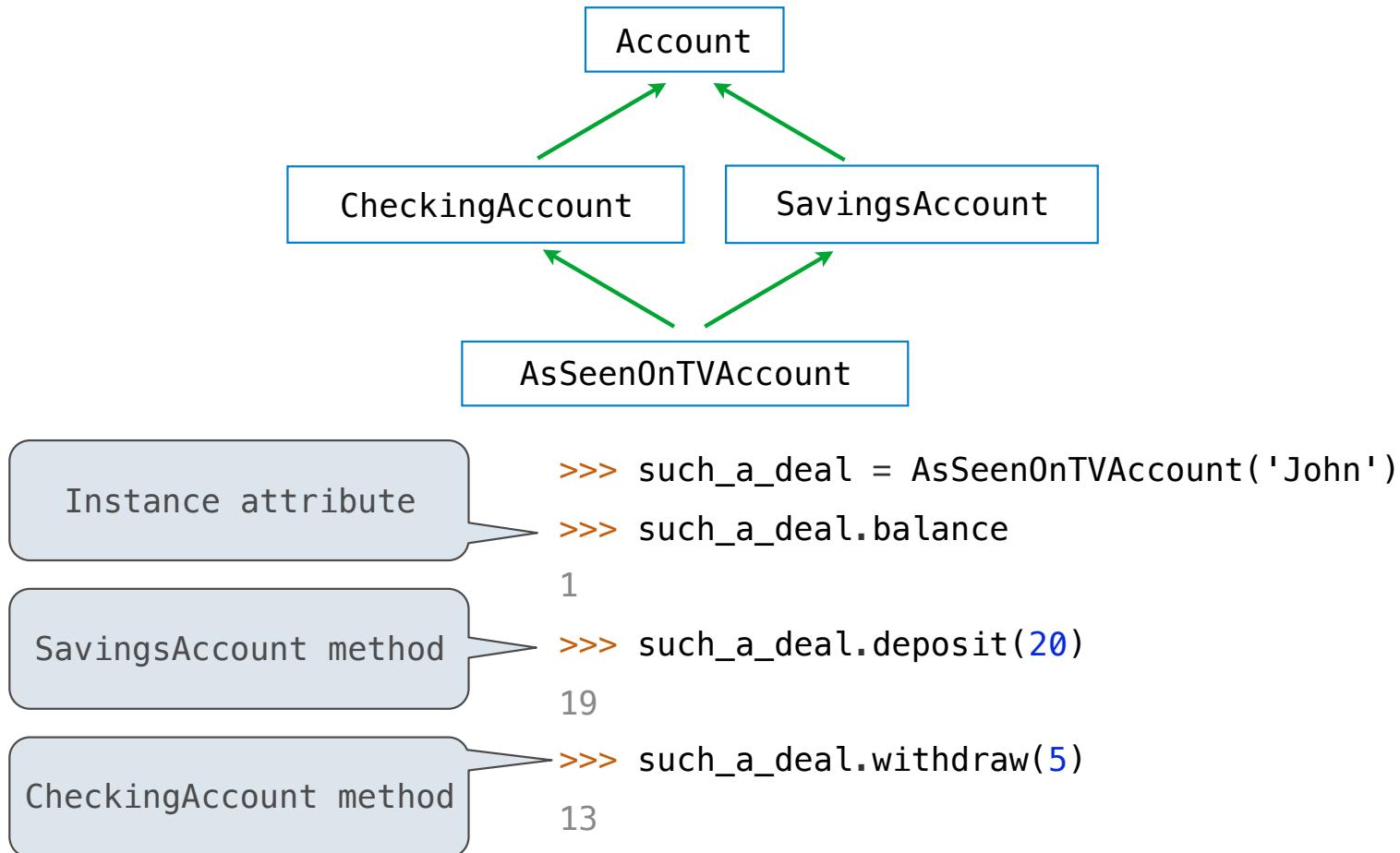
19

CheckingAccount method

```
>>> such_a_deal.withdraw(5)
```

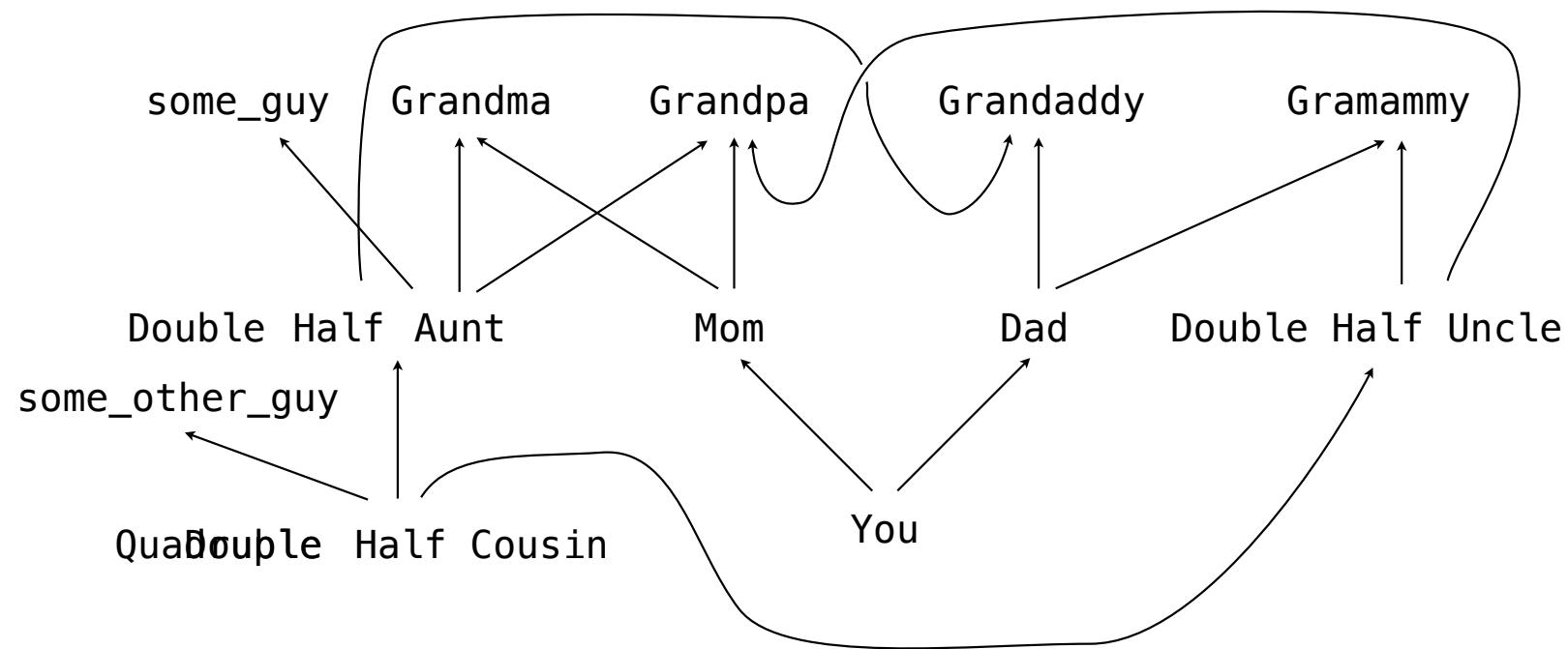
13

Resolving Ambiguous Class Attribute Names



Complicated Inheritance

Biological Inheritance

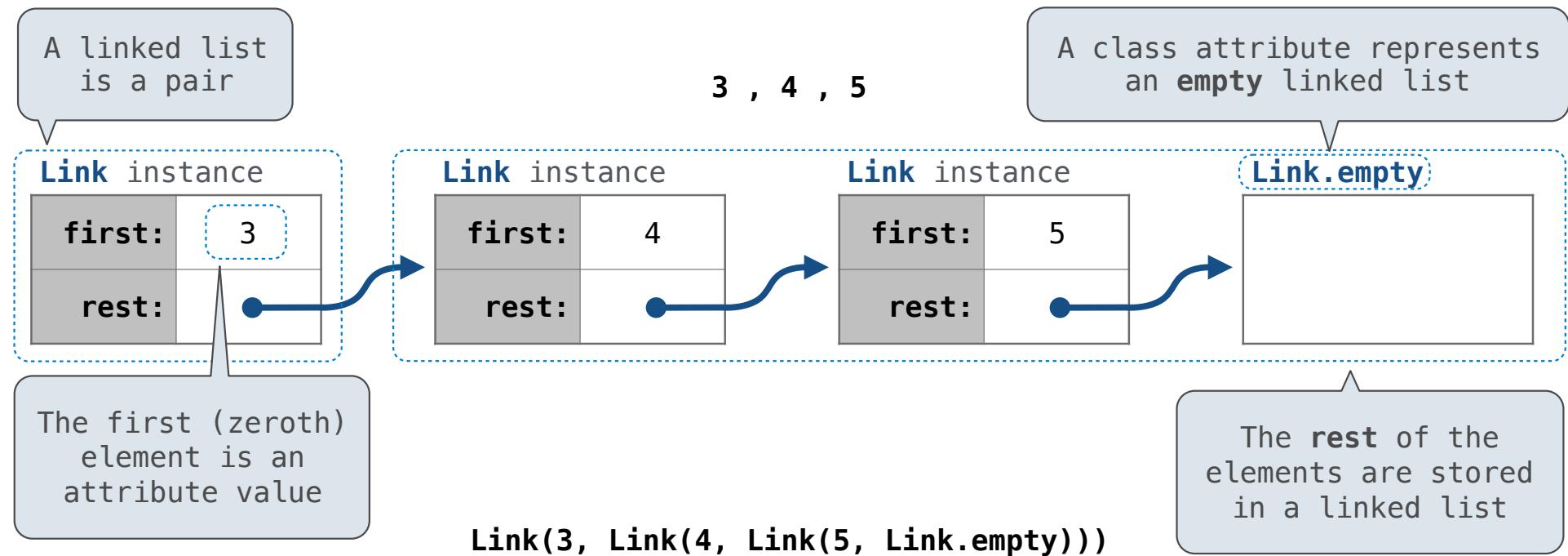


Moral of the story: Inheritance can be complicated, so don't overuse it!

Linked Lists

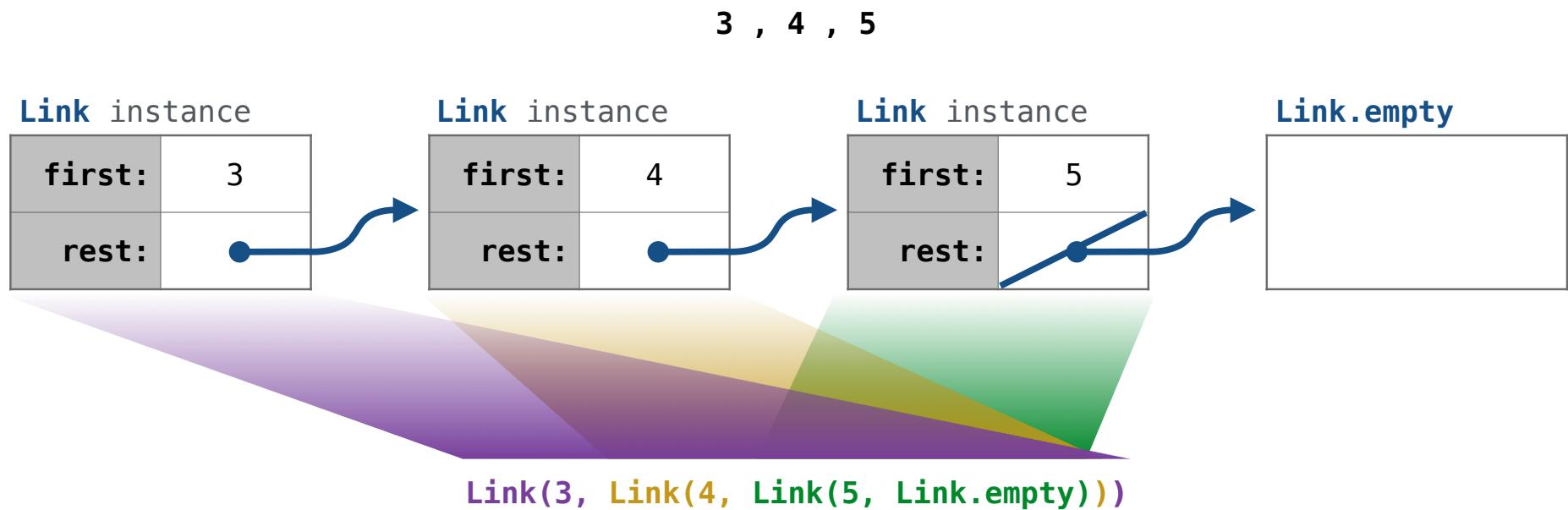
Linked List Structure

A linked list is either empty **or** a first value and the rest of the linked list



Linked List Structure

A linked list is either empty or a first value and the rest of the linked list



Linked List Class

Linked list class: attributes are passed to `__init__`

```
class Link:  
    empty = ()  
    Some zero-length sequence  
  
    def __init__(self, first, rest=empty):  
        assert rest is Link.empty or isinstance(rest, Link)  
        self.first = first  
        self.rest = rest  
        Returns whether  
        rest is a Link
```

`help(isinstance)`: Return whether an object is an instance of a class or of a subclass thereof.

`Link(3, Link(4, Link(5)))`

(Demo)

Property Methods

Property Methods

In some cases, we want the value of instance attributes to be computed on demand

For example, if we want to access the second element of a linked list

```
>>> s = Link(3, Link(4, Link(5)))
>>> s.second
4
>>> s.second = 6
>>> s.second
6
>>> s
Link(3, Link(6, Link(5)))
```

No method calls!

The `@property` decorator on a method designates that it will be called whenever it is looked up on an instance

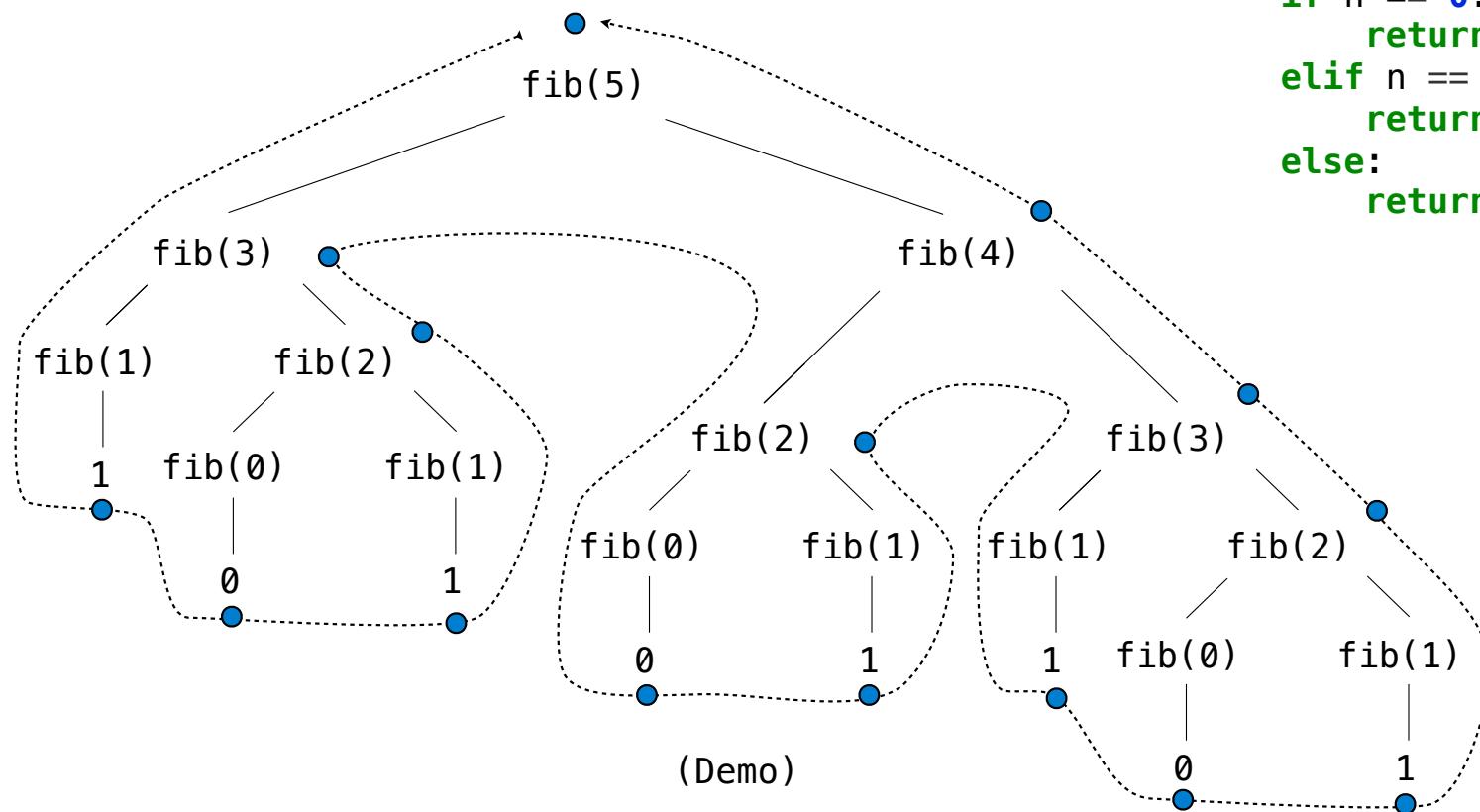
A `@<attribute>.setter` decorator on a method designates that it will be called whenever that attribute is assigned. `<attribute>` must be an existing property method.

(Demo)

Tree Recursion Efficiency

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:



```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```



<http://en.wikipedia.org/wiki/File:Fibonacci.jpg>

Memoization

Memoization

Idea: Remember the results that have been computed before

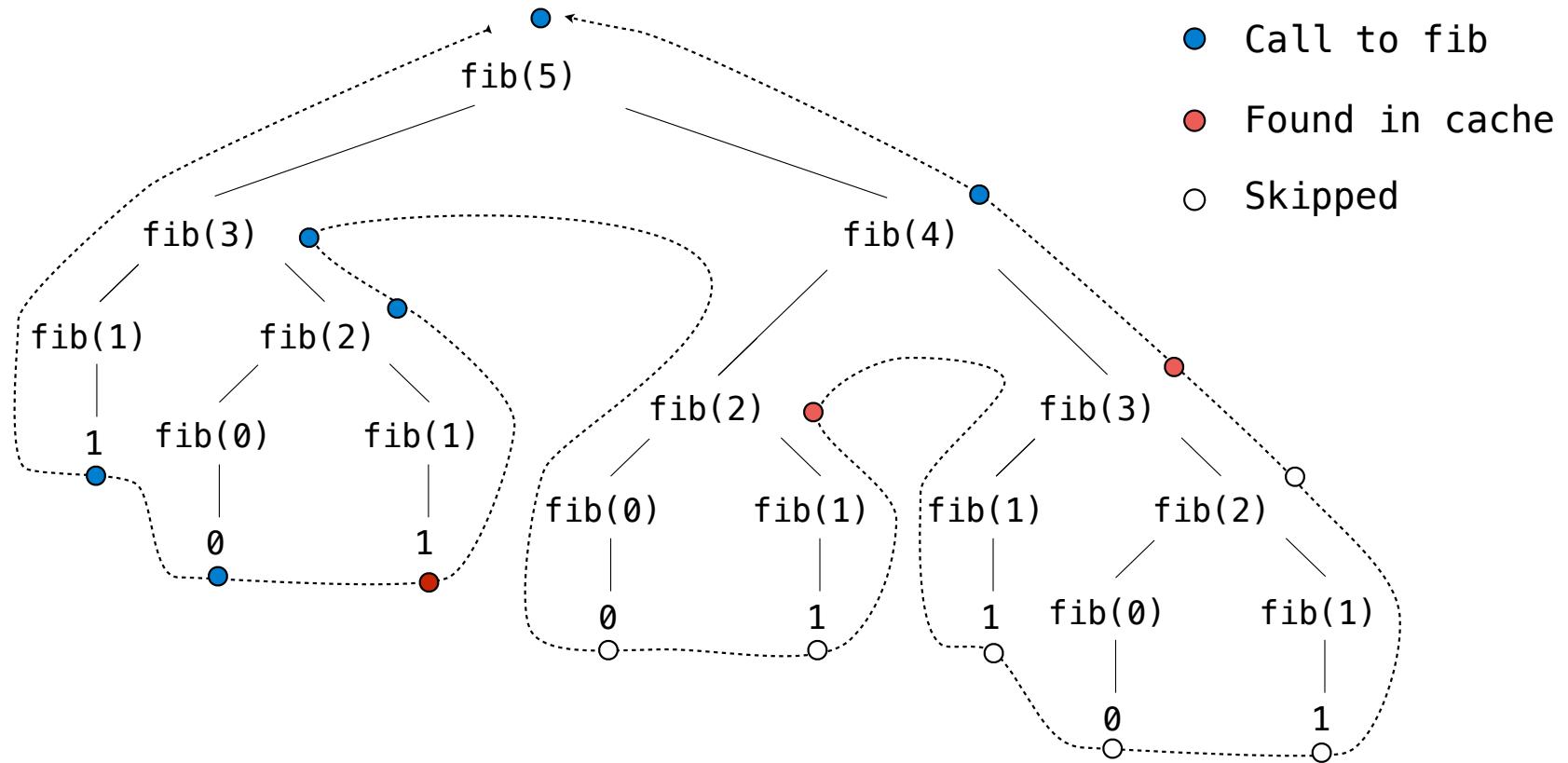
```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

Keys are arguments that map to return values

Same behavior as f, if f is a pure function

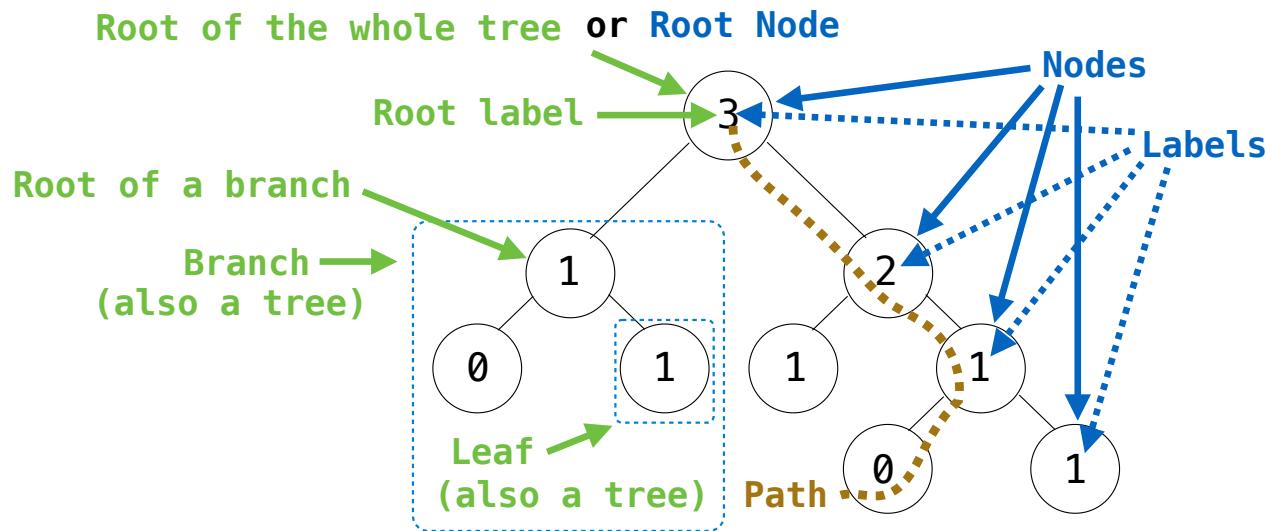
(Demo)

Memoized Tree Recursion



Tree Class

Tree Abstraction (Review)



Recursive description (wooden trees):

A **tree** has a **root label** and a list of **branches**

Each **branch** is a **tree**

A **tree** with zero **branches** is called a **leaf**

A **tree** starts at the **root**

Relative description (family trees):

Each location in a tree is called a **node**

Each **node** has a **label** that can be any value

One node can be the **parent/child** of another

The top node is the **root node**

People often refer to labels by their locations: "each parent is the sum of its children"

Tree Class

A Tree has a label and a list of branches; each branch is a Tree

```
class Tree:  
    def __init__(self, label, branches=[]):  
        self.label = label  
        for branch in branches:  
            assert isinstance(branch, Tree)  
        self.branches = list(branches)  
  
def fib_tree(n):  
    if n == 0 or n == 1:  
        return Tree(n)  
    else:  
        left = fib_tree(n-2)  
        right = fib_tree(n-1)  
        fib_n = left.label + right.label  
        return Tree(fib_n, [left, right])
```

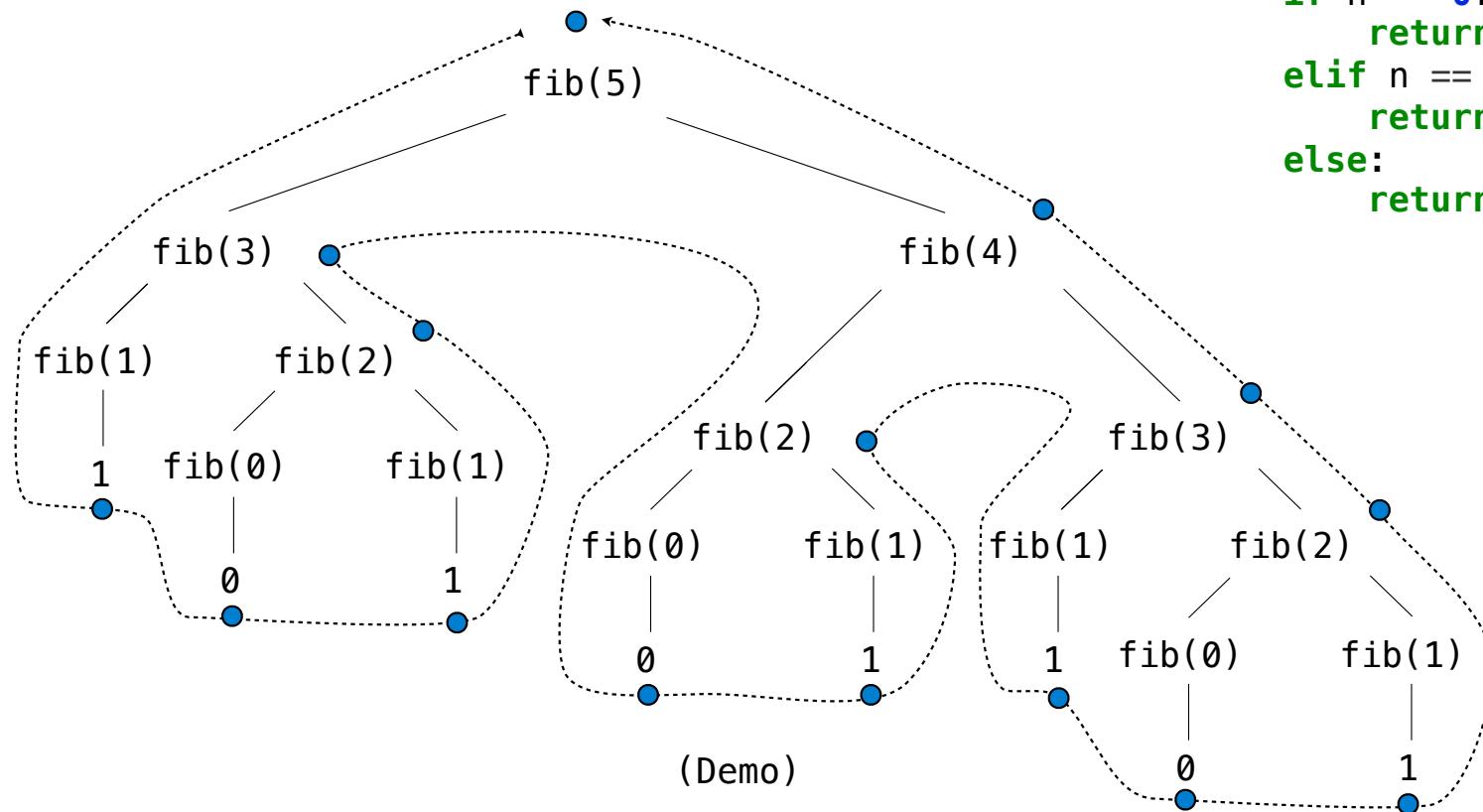
```
def tree(label, branches=[]):  
    for branch in branches:  
        assert is_tree(branch)  
    return [label] + list(branches)  
def label(tree):  
    return tree[0]  
def branches(tree):  
    return tree[1:]  
def fib_tree(n):  
    if n == 0 or n == 1:  
        return tree(n)  
    else:  
        left = fib_tree(n-2)  
        right = fib_tree(n-1)  
        fib_n = label(left) + label(right)  
        return tree(fib_n, [left, right])
```

(Demo)

Measuring Efficiency

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:



```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```



<http://en.wikipedia.org/wiki/File:Fibonacci.jpg>

Memoization

Memoization

Idea: Remember the results that have been computed before

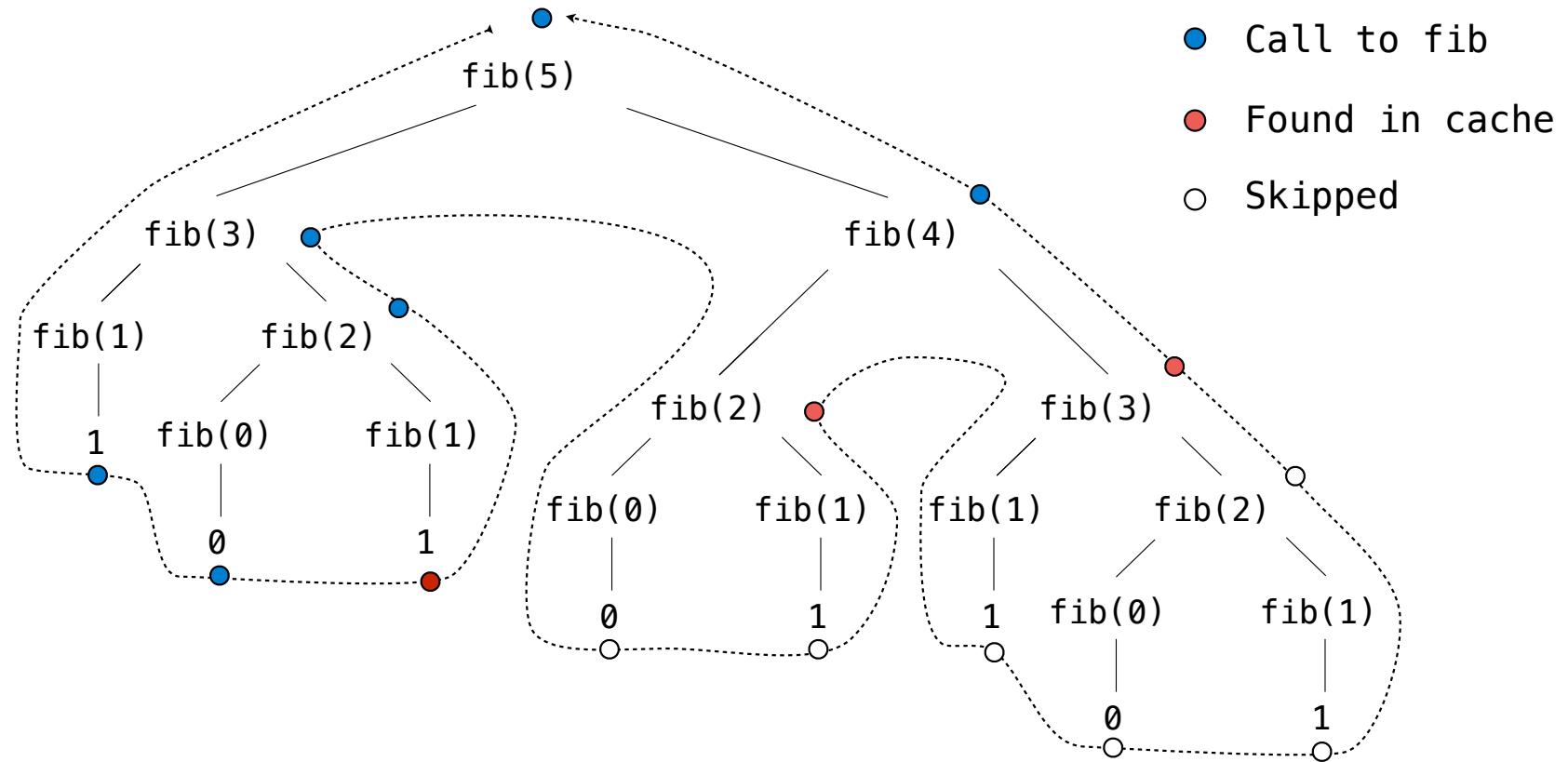
```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

Keys are arguments that map to return values

Same behavior as f, if f is a pure function

(Demo)

Memoized Tree Recursion



Exponentiation

Exponentiation

Goal: one more multiplication lets us double the problem size

```
def exp(b, n):
    if n == 0:
        return 1
    else:
        return b * exp(b, n-1)
```

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{otherwise} \end{cases}$$

```
def exp_fast(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(exp_fast(b, n//2))
    else:
        return b * exp_fast(b, n-1)

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

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ (b^{\frac{1}{2}n})^2 & \text{if } n \text{ is even} \\ b \cdot b^{n-1} & \text{if } n \text{ is odd} \end{cases}$$

(Demo)

Exponentiation

Goal: one more multiplication lets us double the problem size

```
def exp(b, n):
    if n == 0:
        return 1
    else:
        return b * exp(b, n-1)
```

```
def exp_fast(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(exp_fast(b, n//2))
    else:
        return b * exp_fast(b, n-1)

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

Linear time:

- Doubling the input **doubles** the time
- 1024x the input takes 1024x as much time

Logarithmic time:

- Doubling the input **increases** the time by a constant C
- 1024x the input increases the time by only 10 times C

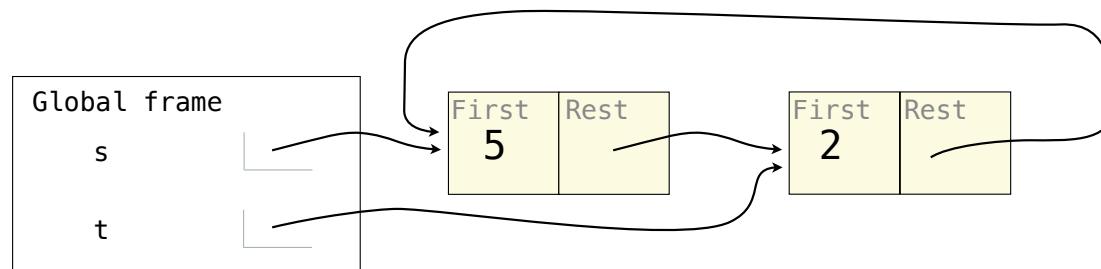
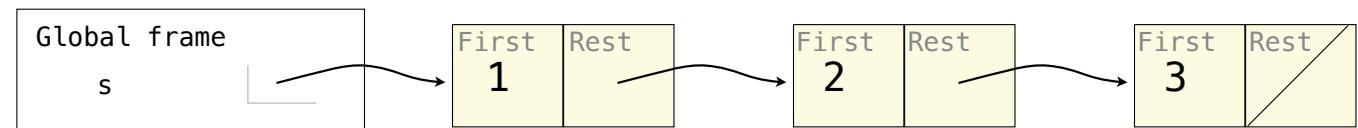
Mutable Linked Lists

Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list

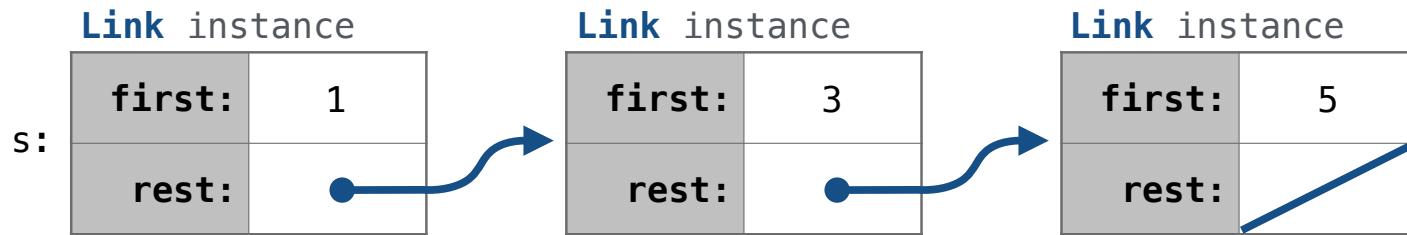
```
>>> s = Link(1, Link(2, Link(3)))
>>> s.first = 5
>>> t = s.rest
>>> t.rest = s
>>> s.first
5
>>> s.rest.rest.rest.rest.first
2
```



Note: The actual environment diagram is much more complicated.

Linked List Mutation Example

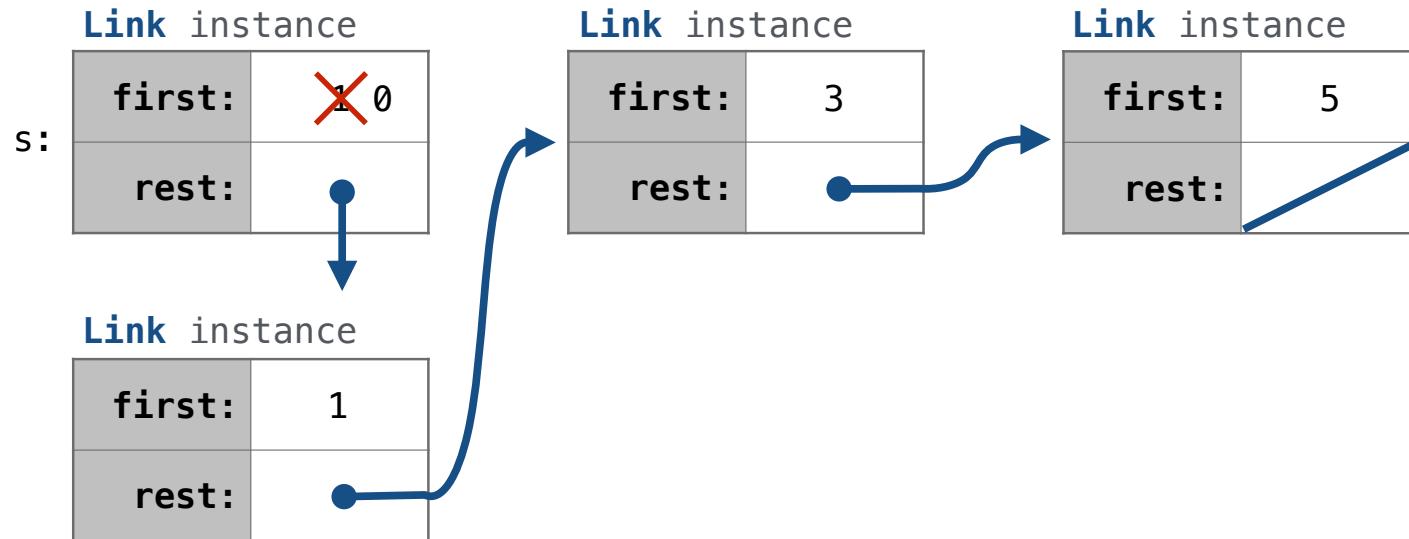
Adding to an Ordered List



```
def add(s, v):
    """Add v to an ordered list s with no repeats, returning modified s."""
    (Note: If v is already in s, then don't modify s, but still return it.)

    add(s, 0)
```

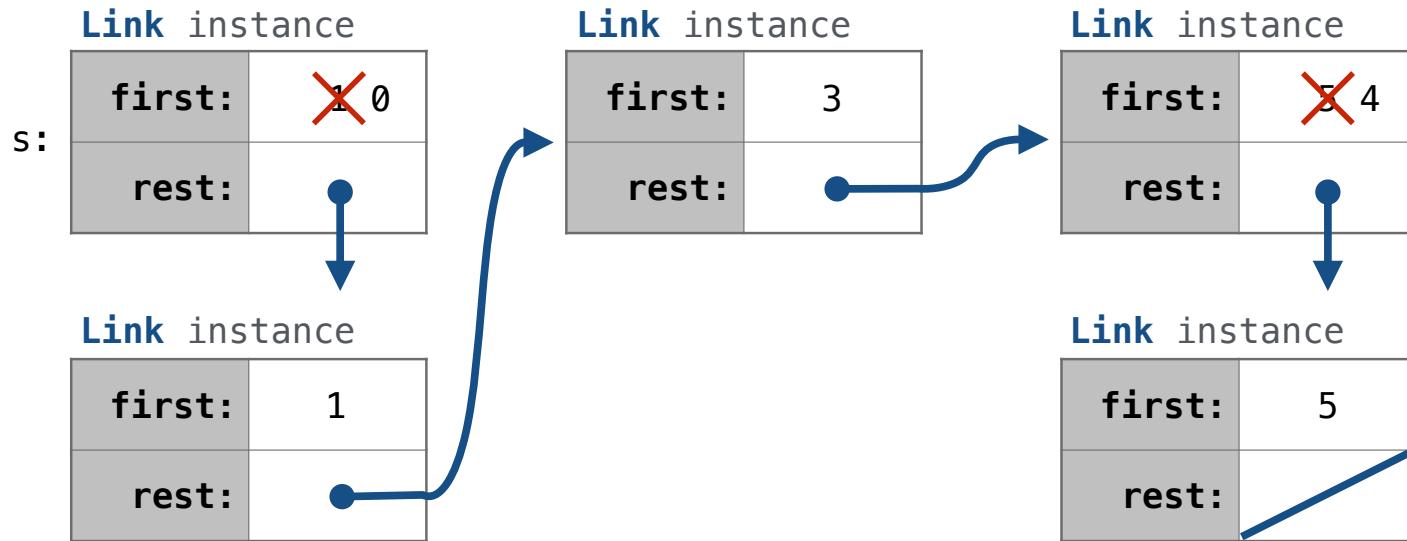
Adding to an Ordered List



```
def add(s, v):
    """Add v to an ordered list s with no repeats, returning modified s."""
    (Note: If v is already in s, then don't modify s, but still return it.)
```

`add(s, 0)` `add(s, 3)` `add(s, 4)`

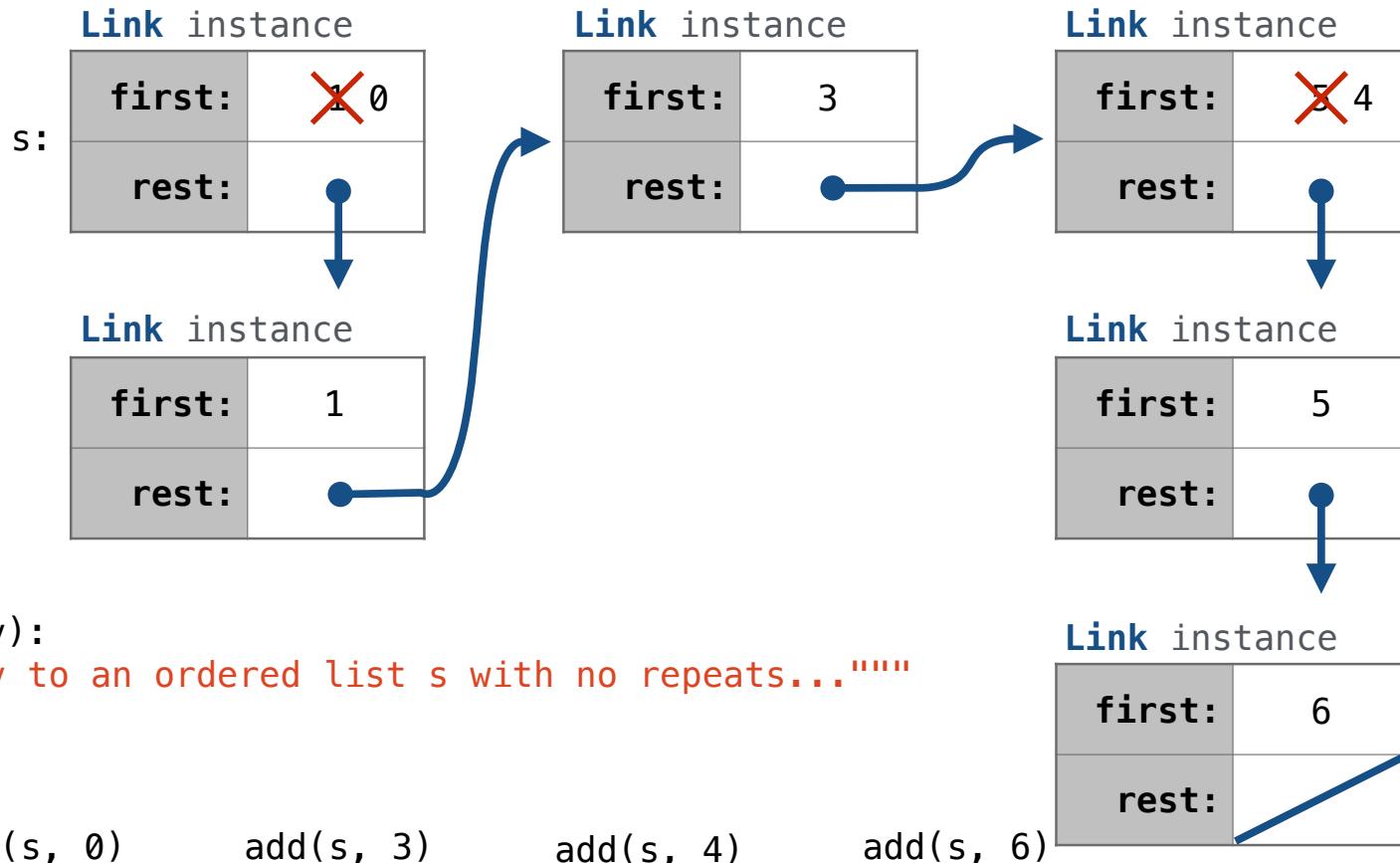
Adding to an Ordered List



```
def add(s, v):
    """Add v to an ordered list s with no repeats..."""
```

add(s, 0) add(s, 3) add(s, 4) add(s, 6)

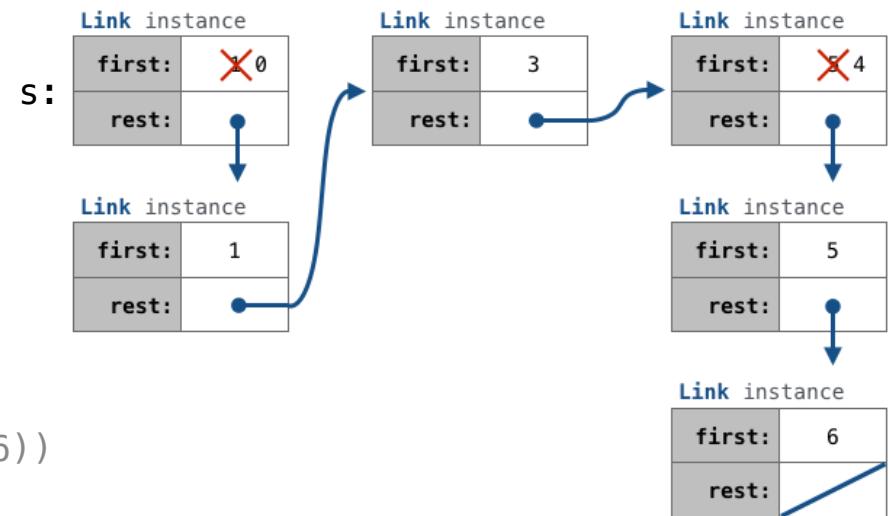
Adding to an Ordered List



Adding to a Set Represented as an Ordered List

```
def add(s, v):
    """Add v to a set s, returning modified s."""

    >>> s = Link(1, Link(3, Link(5)))
    >>> add(s, 0)
    Link(0, Link(1, Link(3, Link(5))))
    >>> add(s, 3)
    Link(0, Link(1, Link(3, Link(5))))
    >>> add(s, 4)
    Link(0, Link(1, Link(3, Link(4, Link(5)))))
    >>> add(s, 6)
    Link(0, Link(1, Link(3, Link(4, Link(5, Link(6))))))
    """
    assert s is not Link.empty
    if s.first > v:
        s.first, s.rest = _____, _____
    elif s.first < v and empty(s.rest):
        s.rest = _____
    elif s.first < v:
        _____
    return s
```

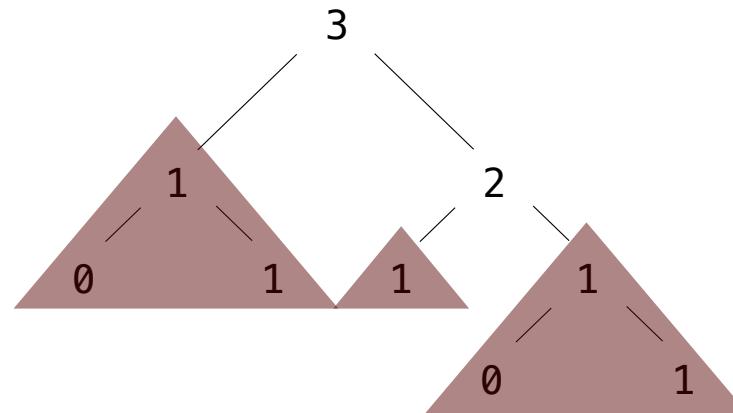


Tree Mutation

Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing



```
def prune(t, n):
    """Prune all sub-trees whose label is n."""
    t.branches = [_____ b _____ for b in t.branches if _____ b.label != n _____]
    for b in t.branches:
        prune(_____ b _____, _____ n _____)
```

String Representations

String Representations

An object value should behave like the kind of data it is meant to represent

For instance, by producing a string representation of itself

Strings are important: they represent language and programs

In Python, all objects produce two string representations:

- The `str` is legible to humans
- The `repr` is legible to the Python interpreter

The `str` and `repr` strings are often the same, but not always

The `repr` String for an Object

The `repr` function returns a Python expression (a string) that evaluates to an equal object

`repr(object)` → string

Return the canonical string representation of the object.
For most object types, `eval(repr(object)) == object`.

The result of calling `repr` on a value is what Python prints in an interactive session

```
>>> 12e12
12000000000000.0
>>> print(repr(12e12))
1200000000000.0
```

Some objects do not have a simple Python-readable string

```
>>> repr(min)
'<built-in function min>'
```

The str String for an Object

Human interpretable strings are useful as well:

```
>>> from fractions import Fraction  
>>> half = Fraction(1, 2)  
>>> repr(half)  
'Fraction(1, 2)'  
>>> str(half)  
'1/2'
```

The result of calling `str` on the value of an expression is what Python prints using the `print` function:

```
>>> print(half)  
1/2
```

(Demo)

Polymorphic Functions

Polymorphic Functions

Polymorphic function: A function that applies to many (poly) different forms (morph) of data

`str` and `repr` are both polymorphic; they apply to any object

`repr` invokes a zero-argument method `__repr__` on its argument

```
>>> half.__repr__()  
'Fraction(1, 2)'
```

`str` invokes a zero-argument method `__str__` on its argument

```
>>> half.__str__()  
'1/2'
```

Implementing repr and str

The behavior of `repr` is slightly more complicated than invoking `__repr__` on its argument:

- An instance attribute called `__repr__` is ignored! Only class attributes are found
- *Question:* How would we implement this behavior?

The behavior of `str` is also complicated:

- An instance attribute called `__str__` is ignored
- If no `__str__` attribute is found, uses `repr` string
- (By the way, `str` is a class, not a function)
- *Question:* How would we implement this behavior?

(Demo)



```
def repr(x):  
    return x.__repr__(x)
```



```
def repr(x):  
    return x.__repr__()
```



```
def repr(x):  
    return type(x).__repr__(x)
```



```
def repr(x):  
    return type(x).__repr__()
```



```
def repr(x):  
    return super(x).__repr__()
```

Interfaces

Message passing: Objects interact by looking up attributes on each other (passing messages)

The attribute look-up rules allow different data types to respond to the same message

A **shared message** (attribute name) that elicits similar behavior from different object classes is a powerful method of abstraction

An interface is a set of shared messages, along with a specification of what they mean

Example:

Classes that implement `__repr__` and `__str__` methods that return Python-interpretable and human-readable strings implement an interface for producing string representations

(Demo)

Special Method Names

Special Method Names in Python

Certain names are special because they have built-in behavior

These names always start and end with two underscores

`__init__` Method invoked automatically when an object is constructed

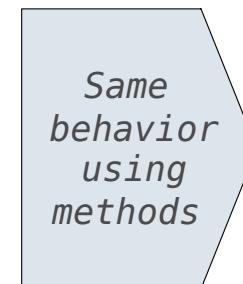
`__repr__` Method invoked to display an object as a Python expression

`__add__` Method invoked to add one object to another

`__bool__` Method invoked to convert an object to True or False

`__float__` Method invoked to convert an object to a float (real number)

```
>>> zero, one, two = 0, 1, 2  
>>> one + two  
3  
>>> bool(zero), bool(one)  
(False, True)
```



```
>>> zero, one, two = 0, 1, 2  
>>> one.__add__(two)  
3  
>>> zero.__bool__(), one.__bool__()  
(False, True)
```

Special Methods

Adding instances of user-defined classes invokes either the `__add__` or `__radd__` method

```
>>> Ratio(1, 3) + Ratio(1, 6)
Ratio(1, 2)
```

```
>>> Ratio(1, 3).__add__(Ratio(1, 6))
Ratio(1, 2)
```

```
>>> Ratio(1, 6).__radd__(Ratio(1, 3))
Ratio(1, 2)
```

<http://getpython3.com/diveintopython3/special-method-names.html>

<http://docs.python.org/py3k/reference/datamodel.html#special-method-names>

(Demo)

Generic Functions

A polymorphic function might take two or more arguments of different types

Type Dispatching: Inspect the type of an argument in order to select behavior

Type Coercion: Convert one value to match the type of another

```
>>> Ratio(1, 3) + 1  
Ratio(4, 3)
```

```
>>> 1 + Ratio(1, 3)  
Ratio(4, 3)
```

```
>>> from math import pi  
>>> Ratio(1, 3) + pi  
3.4749259869231266
```

(Demo)

Announcements

Modular Design

Separation of Concerns

A design principle: Isolate different parts of a program that address different concerns

A modular component can be developed and tested independently

Hog

Hog Game Simulator

- Game rules
- Ordering of events
- State tracking to determine the winner

Game Commentary

- Event descriptions
- State tracking to generate commentary

Player Strategies

- Decision rules
- Strategy parameters (e.g., margins & number of dice)

Ants

Ants Game Simulator

- Order of actions
- Food tracking
- Game ending conditions

Actions

- Characteristics of different ants & bees

Tunnel Structure

- Entrances & exits
- Locations of insects

Example: Restaurant Search

Restaurant Search Data

Given the following data, look up a restaurant by name and show related restaurants.

```
{"business_id": "gclB3ED6uk6viWlolSb_uA", "name": "Cafe 3", "stars": 2.0, "price": 1, ...}  
{"business_id": "WXKx2I2SEzBpeUGtDMCS8A", "name": "La Cascada Taqueria", "stars": 3.0, "price": 2}  
...  
{"business_id": "gclB3ED6uk6viWlolSb_uA", "user_id": "xVocUszkZtAqCxgWak3xVQ", "stars": 1, "text":  
  "Cafe 3 (or Cafe Tre, as I like to say) used to be the bomb diggity when I first lived in the dorms  
  but sadly, quality has dramatically decreased over the years....", "date": "2012-01-19", ...}  
{"business_id": "WXKx2I2SEzBpeUGtDMCS8A", "user_id": "84dCHkhWG8IDtk30VvaY5A", "stars": 2, "text":  
  "-Excuse me for being a snob but if I wanted a room temperature burrito I would take one home,  
  stick it in the fridge for a day, throw it in the microwave for 45 seconds, then eat it. NOT go to  
  a resturant and pay like seven dollars for one...", "date": "2009-04-30", ...}  
...
```

(Demo)

Example: Similar Restaurants

Discussion Question: Most Similar Restaurants

Implement `similar`, a `Restaurant` method that takes a positive integer `k` and a function `similarity` that takes two restaurants as arguments and returns a number. Higher `similarity` values indicate more similar restaurants. The `similar` method returns a list containing the `k` most similar restaurants according to the `similarity` function, but not containing `self`.

```
def similar(self, k, similarity):
    "Return the K most similar restaurants to SELF, using SIMILARITY for comparison."
    others = list(Restaurant.all)
    others.remove(self)
    return sorted(others, key=lambda r: -similarity(self, r))[:k]
```

`sorted`(iterable, /, *, key=None, reverse=False)

Return a new list containing all items from the iterable in ascending order.

A custom key function can be supplied to customize the sort order, and the reverse flag can be set to request the result in descending order.

Example: Reading Files

(Demo)

Set Intersection

Linear-Time Intersection of Sorted Lists

Given two sorted lists with no repeats, return the number of elements that appear in both.



3	4	6	7	9	10
---	---	---	---	---	----



1	3	5	7	8
---	---	---	---	---

(Demo)

```
def fast_overlap(s, t):
    """Return the overlap between sorted S and sorted T.

    >>> fast_overlap([3, 4, 6, 7, 9, 10], [1, 3, 5, 7, 8])
    2
    """
    i, j, count = 0, 0, 0

    while i < len(s) and j < len(t):
        if s[i] == t[j]:
            count, i, j = count + 1, i + 1, j + 1
        elif s[i] < t[j]:
            i = i + 1
        else:
            j = j + 1

    return count
```

Sets

Sets

One more built-in Python container type

- Set literals are enclosed in braces
- Duplicate elements are removed on construction
- Sets have arbitrary order

```
>>> s = {'one', 'two', 'three', 'four', 'four'}
>>> s
{'three', 'one', 'four', 'two'}
>>> 'three' in s
True
>>> len(s)
4
>>> s.union({'one', 'five'})
{'three', 'five', 'one', 'four', 'two'}
>>> s.intersection({'six', 'five', 'four', 'three'})
{'three', 'four'}
>>> s
{'three', 'one', 'four', 'two'}
```

Scheme

Scheme is a Dialect of Lisp

What are people saying about Lisp?

- "If you don't know Lisp, you don't know what it means for a programming language to be powerful and elegant."
 - Richard Stallman, created Emacs & the first free variant of UNIX
- "The only computer language that is beautiful."
 - Neal Stephenson, DeNero's favorite sci-fi author
- "The greatest single programming language ever designed."
 - Alan Kay, co-inventor of Smalltalk and OOP (from the user interface video)

Scheme Expressions

Scheme programs consist of expressions, which can be:

- Primitive expressions: 2 3.3 true + quotient
- Combinations: (quotient 10 2) (not true)

Numbers are self-evaluating; symbols are bound to values

Call expressions include an operator and 0 or more operands in parentheses

```
> (quotient 10 2)  
5  
> (quotient (+ 8 7) 5)  
3  
> (+ (* 3  
      (+ (* 2 4)  
          (+ 3 5)))  
      (- 10 7))  
     6)
```

“quotient” names Scheme’s built-in integer division procedure (i.e., function)

Combinations can span multiple lines (spacing doesn’t matter)

(Demo)

Special Forms

Special Forms

A combination that is not a call expression is a special form:

- **if** expression: (if <predicate> <consequent> <alternative>)
- **and** and **or**: (and <e1> ... <en>), (or <e1> ... <en>)
- Binding symbols: (define <symbol> <expression>)
- New procedures: (define (<symbol> <formal parameters>) <body>)

Evaluation:
(1) Evaluate the predicate expression
(2) Evaluate either the consequent or alternative

```
> (define pi 3.14)
> (* pi 2)
6.28
```

The symbol “pi” is bound to 3.14 in the global frame

```
> (define (abs x)
  (if (< x 0)
      (- x)
      x))
> (abs -3)
3
```

A procedure is created and bound to the symbol “abs”

(Demo)

Scheme Interpreters

(Demo)

Lambda Expressions

Lambda Expressions

Lambda expressions evaluate to anonymous procedures

(lambda (<formal-parameters>) <body>)



Two equivalent expressions:

(define (plus4 x) (+ x 4))

(define plus4 (lambda (x) (+ x 4)))

An operator can be a call expression too:

(lambda (x y z) (+ x y (square z))) 1 2 3) ➔ 12

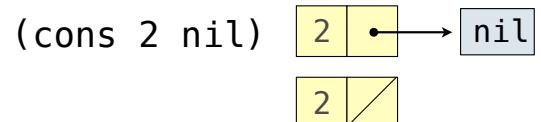
Evaluates to the
 $x+y+z^2$ procedure

Lists

Scheme Lists

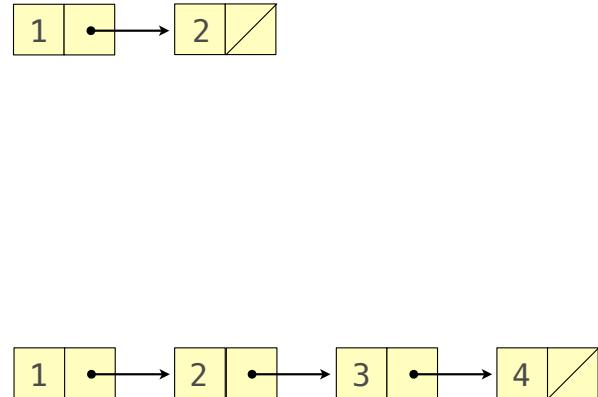
In the late 1950s, computer scientists used confusing names

- **cons**: Two-argument procedure that creates a linked list
- **car**: Procedure that returns the first element of a list
- **cdr**: Procedure that returns the rest of a list
- **nil**: The empty list



Important! Scheme lists are written in parentheses with elements separated by spaces

```
> (cons 1 (cons 2 nil))
(1 2)
> (define x (cons 1 (cons 2 nil)))
> x
(1 2)
> (car x)
1
> (cdr x)
(2)
> (cons 1 (cons 2 (cons 3 (cons 4 nil))))
(1 2 3 4)
```



(Demo)

Symbolic Programming

Symbolic Programming

Symbols normally refer to values; how do we refer to symbols?

```
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
```

No sign of "a" and "b" in the resulting value

Quotation is used to refer to symbols directly in Lisp.

```
> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)
```

Short for (quote a), (quote b):
Special form to indicate that the expression itself is the value.

Quotation can also be applied to combinations to form lists.

```
> '(a b c)
(a b c)
> (car '(a b c))
a
> (cdr '(a b c))
(b c)
```

(Demo)

Pairs Review

Pairs and Lists

In the late 1950s, computer scientists used confusing names

- **cons**: Two-argument procedure that creates a pair
- **car**: Procedure that returns the first element of a pair
- **cdr**: Procedure that returns the second element of a pair
- **nil**: The empty list
- A (non-empty) list in Scheme is a pair in which the second element is **nil** or a Scheme list
- **Important!** Scheme lists are written in parentheses separated by spaces
- A dotted list has some value for the second element of the last pair that is not a list

```
> (cons 1 (cons 2 nil))  
(1 2)  
> (define x (cons 1 2))  
> x  
(1 . 2)  
> (car x)          Not a well-formed list!  
1  
> (cdr x)  
2  
> (cons 1 (cons 2 (cons 3 (cons 4 nil))))  
(1 2 3 4)
```

(Demo)

Sierpinski's Triangle

(Demo)

Programming Languages

Programming Languages

A computer typically executes programs written in many different programming languages

Machine languages: statements are interpreted by the hardware itself

- A fixed set of instructions invoke operations implemented by the circuitry of the central processing unit (CPU)
- Operations refer to specific hardware memory addresses; no abstraction mechanisms

High-level languages: statements & expressions are interpreted by another program or compiled (translated) into another language

- Provide means of abstraction such as naming, function definition, and objects
- Abstract away system details to be independent of hardware and operating system

Python 3

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

```
from dis import dis  
dis(square)
```

Python 3 Byte Code

LOAD_FAST	0 (x)
LOAD_FAST	0 (x)
BINARY_MULTIPLY	
RETURN_VALUE	

Metalinguistic Abstraction

A powerful form of abstraction is to define a new language that is tailored to a particular type of application or problem domain

Type of application: Erlang was designed for concurrent programs. It has built-in elements for expressing concurrent communication. It is used, for example, to implement chat servers with many simultaneous connections

Problem domain: The MediaWiki mark-up language was designed for generating static web pages. It has built-in elements for text formatting and cross-page linking. It is used, for example, to create Wikipedia pages

A programming language has:

- **Syntax:** The legal statements and expressions in the language
- **Semantics:** The execution/evaluation rule for those statements and expressions

To create a new programming language, you either need a:

- **Specification:** A document describe the precise syntax and semantics of the language
- **Canonical Implementation:** An interpreter or compiler for the language

Parsing

Reading Scheme Lists

A Scheme list is written as elements in parentheses:

(`<element_0>` `<element_1>` ... `<element_n>`) A Scheme list

Each `<element>` can be a combination or primitive

(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))

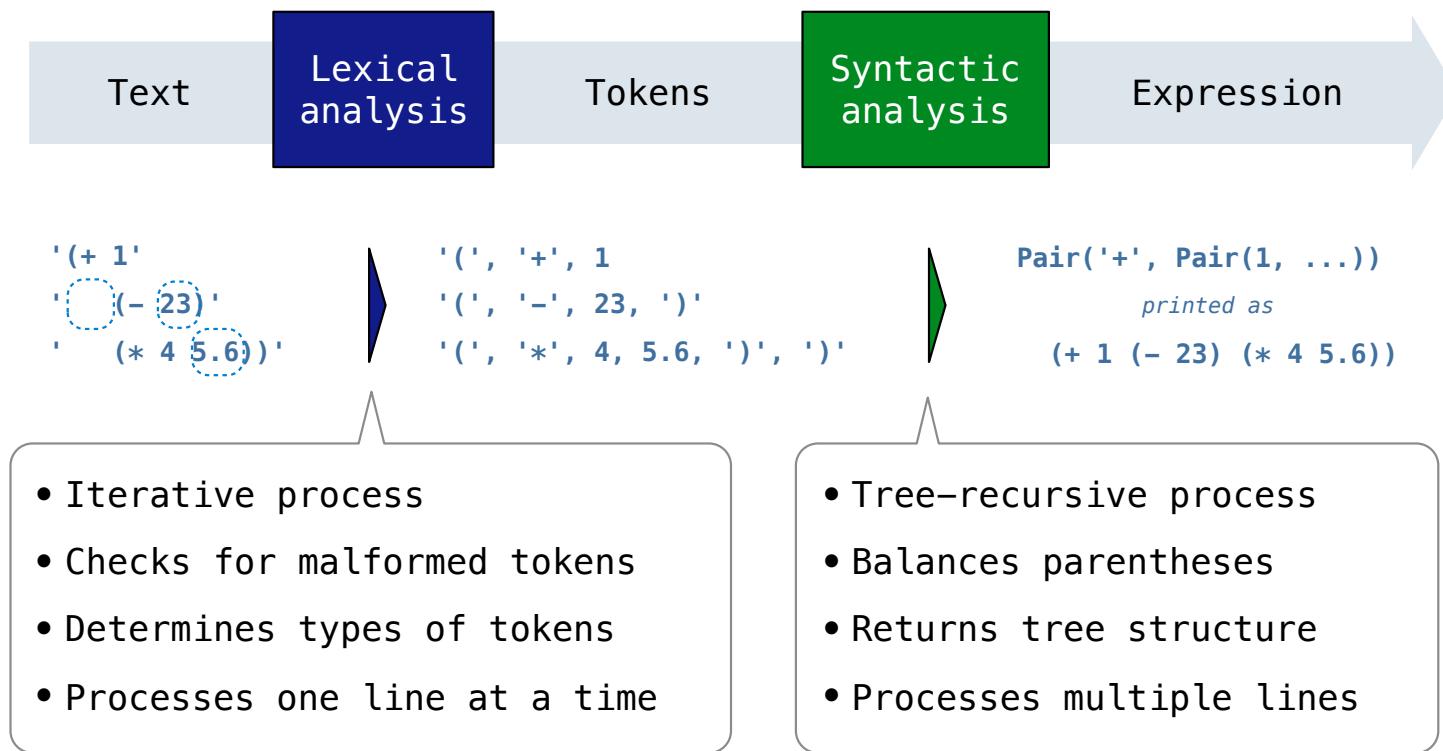
The task of parsing a language involves coercing a string representation of an expression to the expression itself

(Demo)

http://composingprograms.com/examples/scalc/scheme_reader.py.html

Parsing

A Parser takes text and returns an expression



Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested

Each call to scheme_read consumes the input tokens for exactly one expression

'(', '+', 1, '(', '-', 23, ')', '(', '*', 4, 5.6, ')', ')'
▲

Base case: symbols and numbers

Recursive call: scheme_read sub-expressions and combine them

(Demo)

Scheme-Syntax Calculator

(Demo)

The Pair Class

The Pair class represents Scheme pairs and lists. A list is a pair whose second element is either a list or nil.

```
class Pair:  
    """A Pair has two instance attributes:  
    first and second.  
    """
```

*For a Pair to be a well-formed list,
second is either a well-formed list or nil.
Some methods only apply to well-formed lists.*

```
def __init__(self, first, second):  
    self.first = first  
    self.second = second
```

```
>>> s = Pair(1, Pair(2, Pair(3, nil)))  
>>> print(s)  
(1 2 3)  
>>> len(s)  
3  
>>> print(Pair(1, 2))  
(1 . 2)  
>>> print(Pair(1, Pair(2, 3)))  
(1 2 . 3)  
>>> len(Pair(1, Pair(2, 3)))  
Traceback (most recent call last):  
...  
TypeError: length attempted on improper list
```

Scheme expressions are represented as Scheme lists! Source code is data

(Demo)

Calculator Syntax

The Calculator language has primitive expressions and call expressions. (That's it!)

A primitive expression is a number: 2 -4 5.6

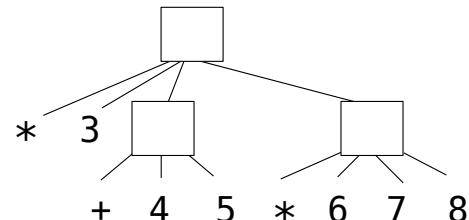
A call expression is a combination that begins with an operator (+, -, *, /) followed by 0 or more expressions: (+ 1 2 3) (/ 3 (+ 4 5))

Expressions are represented as Scheme lists (Pair instances) that encode tree structures.

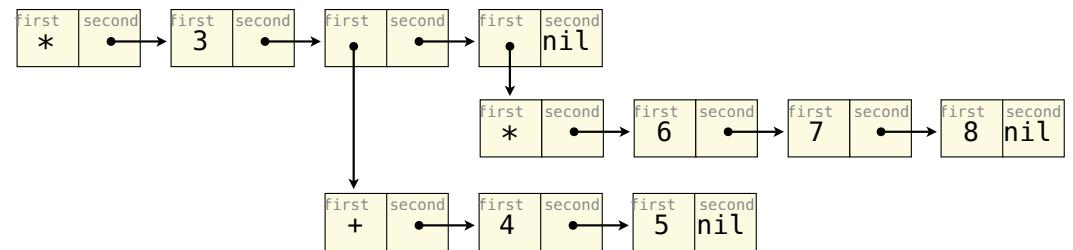
Expression

```
(* 3  
  (+ 4 5)  
  (* 6 7 8))
```

Expression Tree



Representation as Pairs



Calculator Semantics

The value of a calculator expression is defined recursively.

Primitive: A number evaluates to itself.

Call: A call expression evaluates to its argument values combined by an operator.

+: Sum of the arguments

*****: Product of the arguments

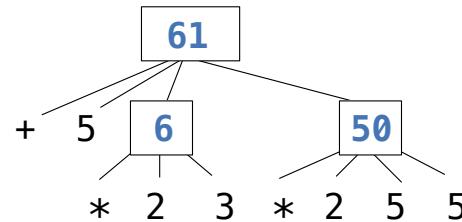
-: If one argument, negate it. If more than one, subtract the rest from the first.

/: If one argument, invert it. If more than one, divide the rest from the first.

Expression

```
(+ 5  
  (* 2 3)  
  (* 2 5 5))
```

Expression Tree



Evaluation

The Eval Function

The eval function computes the value of an expression, which is always a number

It is a generic function that dispatches on the type of the expression (primitive or call)

Implementation

```
def calc_eval(exp):
    if type(exp) in (int, float):
        return exp
    elif isinstance(exp, Pair):
        arguments = exp.second.map(calc_eval)
        return calc_apply(exp.first, arguments)
    else:
        raise TypeError
```

Recursive call
returns a number
for each operand

calc_apply
'+', '-'
'*', '/'

A Scheme list
of numbers

Language Semantics

A number evaluates...

to itself

A call expression evaluates...

*to its argument values
combined by an operator*

Applying Built-in Operators

The apply function applies some operation to a (Scheme) list of argument values

In calculator, all operations are named by built-in operators: +, -, *, /

Implementation

```
def calc_apply(operator, args):
    if operator == '+':
        return reduce(add, args, 0)
    elif operator == '-':
        ...
    elif operator == '*':
        ...
    elif operator == '/':
        ...
    else:
        raise TypeError
```

Language Semantics

+:	<i>Sum of the arguments</i>
-:	...
*:	...
/:	...

(Demo)

Interactive Interpreters

Read-Eval-Print Loop

The user interface for many programming languages is an interactive interpreter

1. Print a prompt
2. **Read** text input from the user
3. Parse the text input into an expression
4. **Evaluate** the expression
5. If any errors occur, report those errors, otherwise
6. **Print** the value of the expression and repeat

(Demo)

Raising Exceptions

Exceptions are raised within lexical analysis, syntactic analysis, eval, and apply

Example exceptions

- **Lexical analysis:** The token 2.3.4 raises ValueError("invalid numeral")
- **Syntactic analysis:** An extra) raises SyntaxError("unexpected token")
- **Eval:** An empty combination raises TypeError("() is not a number or call expression")
- **Apply:** No arguments to - raises TypeError("- requires at least 1 argument")

(Demo)

Handling Exceptions

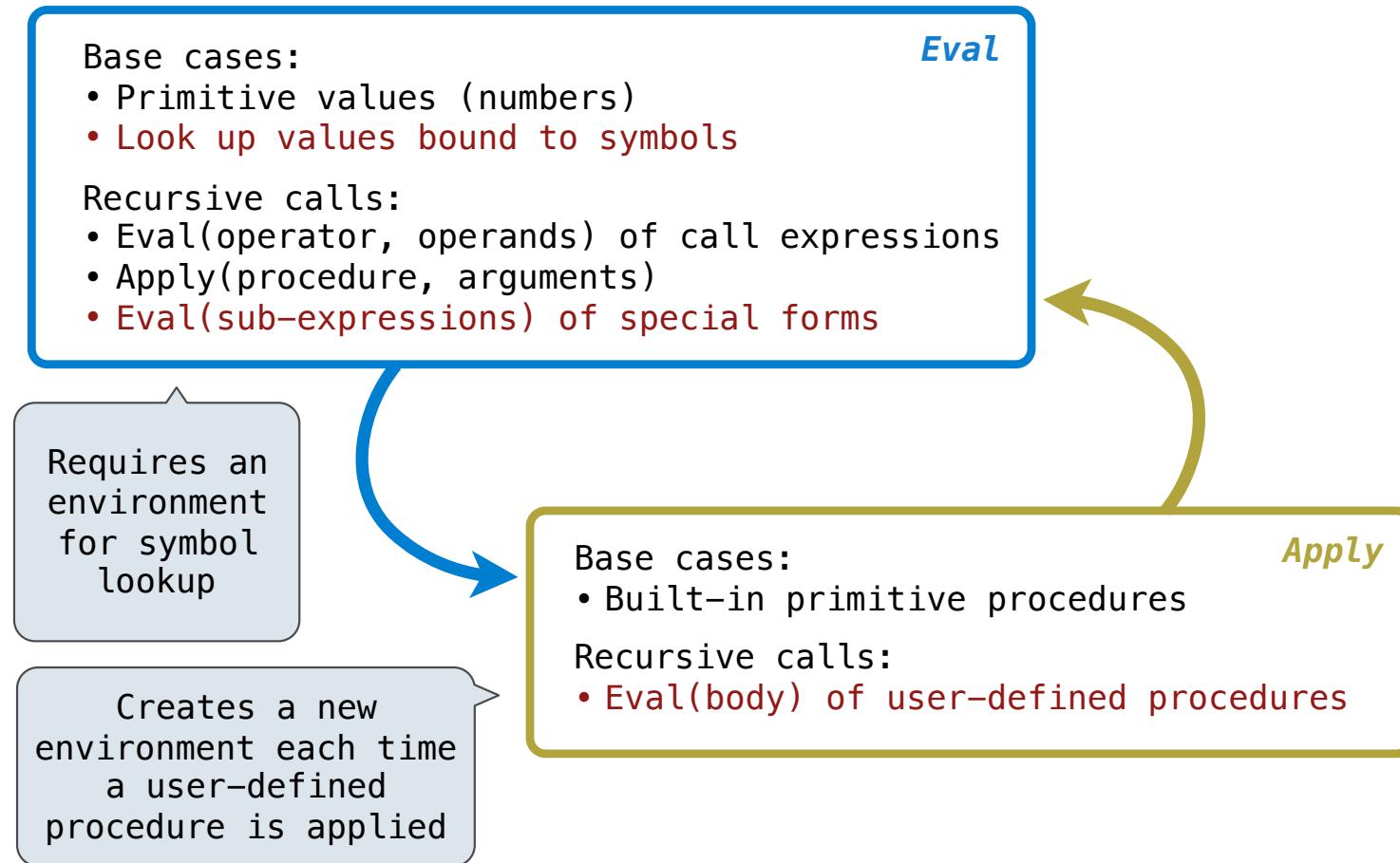
An interactive interpreter prints information about each error

A well-designed interactive interpreter should not halt completely on an error,
so that the user has an opportunity to try again in the current environment

(Demo)

Interpreting Scheme

The Structure of an Interpreter

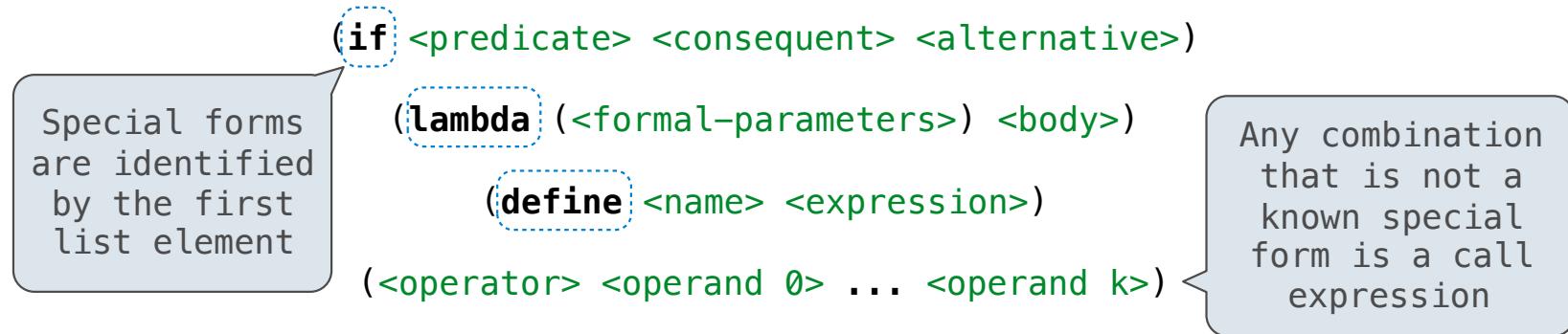


Special Forms

Scheme Evaluation

The scheme_eval function choose behavior based on expression form:

- Symbols are looked up in the current environment
- Self-evaluating expressions are returned as values
- All other legal expressions are represented as Scheme lists, called combinations



```
(define (demo s) (if (null? s) '(3) (cons (car s) (demo (cdr s)))))  
                      (demo (list 1 2)))
```

Logical Forms

Logical Special Forms

Logical forms may only evaluate some sub-expressions

- **If** expression: (**if** <predicate> <consequent> <alternative>)
- **And** and **or**: (**and** <e1> ... <en>), (**or** <e1> ... <en>)
- **Cond** expression: (**cond** (<p1> <e1>) ... (<pn> <en>) (else <e>))

The value of an if expression is the value of a sub-expression:

- Evaluate the predicate
- Choose a sub-expression: <consequent> or <alternative>
- Evaluate that sub-expression to get the value of the whole expression

do_if_form

(Demo)

Quotation

Quotation

The quote special form evaluates to the quoted expression, which is not evaluated

(quote <expression>)

(quote (+ 1 2))

evaluates to the
three-element Scheme list

(+ 1 2)

The <expression> itself is the value of the whole quote expression

'<expression> is shorthand for (quote <expression>)

(quote (1 2))

is equivalent to

'(1 2)

The scheme_read parser converts shorthand ' to a combination that starts with quote

(Demo)

Lambda Expressions

Lambda Expressions

Lambda expressions evaluate to user-defined procedures

```
(lambda (<formal-parameters>) <body>)
```

```
(lambda (x) (* x x))
```

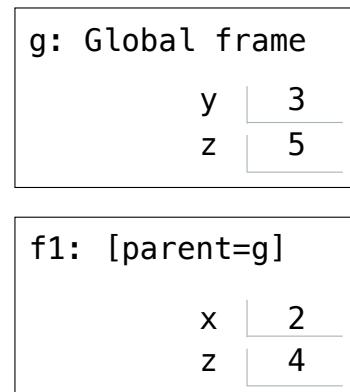
```
class LambdaProcedure:  
    def __init__(self, formals, body, env):  
        self.formals = formals ..... A scheme list of symbols  
        self.body = body ..... A scheme list of expressions  
        self.env = env ..... A Frame instance
```

Frames and Environments

A frame represents an environment by having a parent frame

Frames are Python instances with methods `lookup` and `define`

In Project 4, Frames do not hold return values



(Demo)

Define Expressions

Define Expressions

Define binds a symbol to a value in the first frame of the current environment.

```
(define <name> <expression>)
```

1. Evaluate the <expression>
2. Bind <name> to its value in the current frame

```
(define x (+ 1 2))
```

Procedure definition is shorthand of define with a lambda expression

```
(define (<name> <formal parameters>) <body>)
```

```
(define <name> (lambda (<formal parameters>) <body>))
```

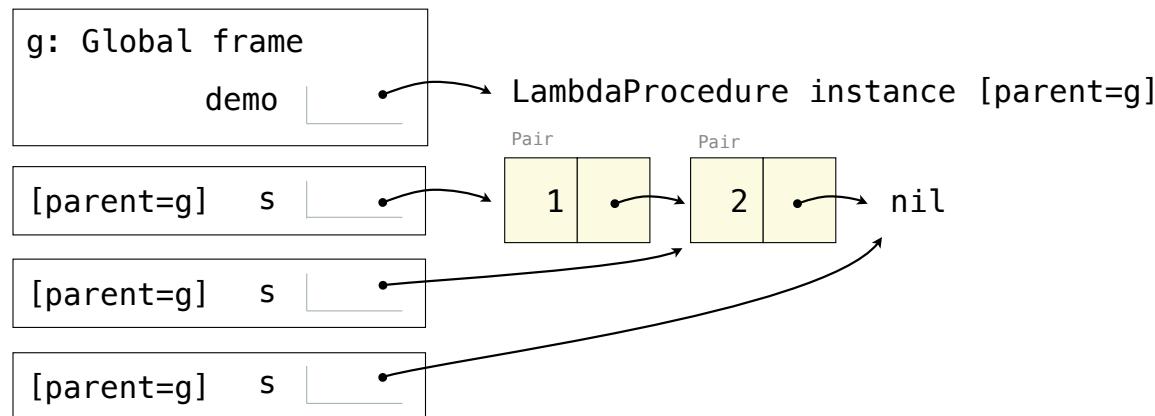
Applying User-Defined Procedures

To apply a user-defined procedure, create a new frame in which formal parameters are bound to argument values, whose parent is the `env` attribute of the procedure

Evaluate the body of the procedure in the environment that starts with this new frame

```
(define (demo s) (if (null? s) '(3) (cons (car s) (demo (cdr s)))))

(demo (list 1 2))
```



Eval/Apply in Lisp 1.5

```
apply[fn;x;a] =  
  [atom[fn] → [eq[fn;CAR] → caar[x];  
               eq[fn;CDR] → cdar[x];  
               eq[fn;CONS] → cons[car[x];cadr[x]];  
               eq[fn;ATOM] → atom[car[x]]];  
               eq[fn;EQ] → eq[car[x];cadr[x]];  
               T → apply[eval[fn;a];x;a]];  
  eq[car[fn];LAMBDA] → eval[caddr[fn];pairlis[cadr[fn];x;a]];  
  eq[car[fn];LABEL] → apply[caddr[fn];x;cons[cons[cadr[fn];  
                                                caddr[fn]];a]]]  
  
eval[e;a] = [atom[e] → cdr[assoc[e;a]]];  
            atom[car[e]] →  
              [eq[car[e],QUOTE] → cadr[e];  
               eq[car[e];COND] → evcon[cdr[e];a];  
               T → apply[car[e];evlis[cdr[e];a];a]];  
            T → apply[car[e];evlis[cdr[e];a];a]]
```

Dynamic Scope

Dynamic Scope

The way in which names are looked up in Scheme and Python is called lexical scope (or static scope) [You can see what names are in scope by inspecting the definition]

Lexical scope: The parent of a frame is the environment in which a procedure was *defined*

Dynamic scope: The parent of a frame is the environment in which a procedure was *called*

Special form to create dynamically scoped procedures (**mu** special form only exists in Project 4 Scheme)

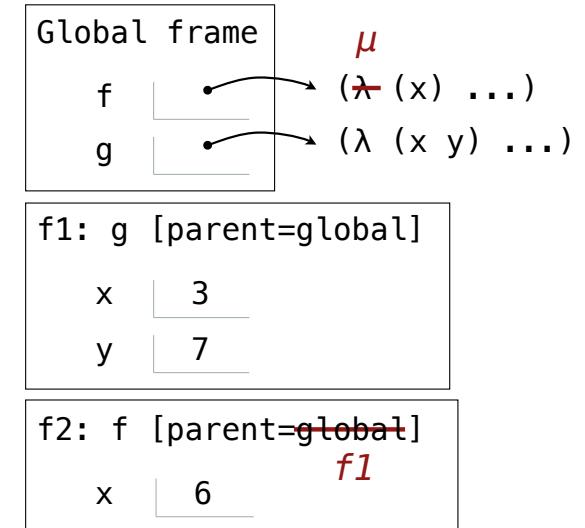
(define f (lambda (x) (+ x y)))
(define g (lambda (x y) (f (+ x x))))
(g 3 7)

Lexical scope: The parent for f's frame is the global frame

Error: unknown identifier: y

Dynamic scope: The parent for f's frame is g's frame

13



Tail Recursion

Functional Programming

All functions are pure functions

No re-assignment and no mutable data types

Name-value bindings are permanent

Advantages of functional programming:

- The value of an expression is independent of the order in which sub-expressions are evaluated
- Sub-expressions can safely be evaluated in parallel or only on demand (lazily)
- **Referential transparency:** The value of an expression does not change when we substitute one of its subexpression with the value of that subexpression

But... no `for/while` statements! Can we make basic iteration efficient? Yes!

Recursion and Iteration in Python

In Python, recursive calls always create new active frames

`factorial(n, k)` computes: $n! * k$

Time	Space
$\Theta(n)$	$\Theta(n)$
$\Theta(n)$	$\Theta(1)$

```
def factorial(n, k):
    if n == 0:
        return k
    else:
        return factorial(n-1, k*n)

def factorial(n, k):
    while n > 0:
        n, k = n-1, k*n
    return k
```

Tail Recursion

From the Revised⁷ Report on the Algorithmic Language Scheme:

"Implementations of Scheme are required to be properly tail-recursive. This allows the execution of an iterative computation in constant space, even if the iterative computation is described by a syntactically recursive procedure."

```
(define (factorial n k)
  (if (zero? n) k
      (factorial (- n 1)
                 (* k n))))
```

Should use resources like

```
def factorial(n, k):
    while n > 0:
        n, k = n-1, k*n
    return k
```

How? Eliminate the middleman!

Time Space

$\Theta(n)$ $\Theta(1)$

(Demo)

[Interactive Diagram](#)

Tail Calls

Tail Calls

A procedure call that has not yet returned is **active**. Some procedure calls are **tail calls**. A Scheme interpreter should support an **unbounded number** of active tail calls using only a **constant** amount of space.

A tail call is a call expression in a tail context:

- The last body sub-expression in a **lambda** expression
- Sub-expressions 2 & 3 in a tail context **if** expression
- All non-predicate sub-expressions in a tail context **cond**
- The last sub-expression in a tail context **and**, **or**, **begin**, or **let**

```
(define (factorial n k)
  (if (= n 0) k
      (factorial (- n 1)
                 (* k n)))))
```

Example: Length of a List

```
(define (length s)
  (if (null? s) 0
      (+ 1 (length (cdr s)))))
```

The code is shown with a blue dashed box around the entire body of the function. Inside this box, the call expression `(length (cdr s))` is highlighted with a red rounded rectangle. A grey speech bubble above it contains the text "Not a tail context".

A call expression is not a tail call if more computation is still required in the calling procedure

Linear recursive procedures can often be re-written to use tail calls

```
(define (length-tail s)
  (define (length-iter s n)
    (if (null? s) n
        (length-iter (cdr s) (+ 1 n)))))
```

The code is shown with a blue dashed box around the entire body of the `length-iter` function. Inside this box, the recursive call `(length-iter (cdr s) (+ 1 n))` is highlighted with a blue rounded rectangle. A grey speech bubble above it contains the text "Recursive call is a tail call".

[Eval with Tail Call Optimization](#)

The return value of the tail call is the return value of the current procedure call
Therefore, tail calls shouldn't increase the environment size

(Demo)

Tail Recursion Examples

Which Procedures are Tail Recursive?

Which of the following procedures run in constant space? $\Theta(1)$

`;; Compute the length of s.`

```
(define (length s)
  (+ 1 (if (null? s)
            -1
            (length (cdr s)))))
```

`;; Return the nth Fibonacci number.`

```
(define (fib n)
  (define (fib-iter current k)
    (if (= k n)
        current
        (fib-iter (+ current
                      (fib (- k 1)))
                  (+ k 1)))))
```

`(if (= 1 n) 0 (fib-iter 1 2)))`

`;; Return whether s contains v.`

```
(define (contains s v)
  (if (null? s)
      false
      (if (= v (car s))
          true
          (contains (cdr s) v))))
```

`;; Return whether s has any repeated elements.`

```
(define (has-repeat s)
  (if (null? s)
      false
      (if (contains? (cdr s) (car s))
          true
          (has-repeat (cdr s)))))
```

Map and Reduce

Example: Reduce

```
(define (reduce procedure s start)
  (if (null? s) start
      (reduce procedure
              (cdr s)
              (procedure start (car s))))) ) )
```

Recursive call is a tail call

Space depends on what procedure requires

(reduce * '(3 4 5) 2)

120

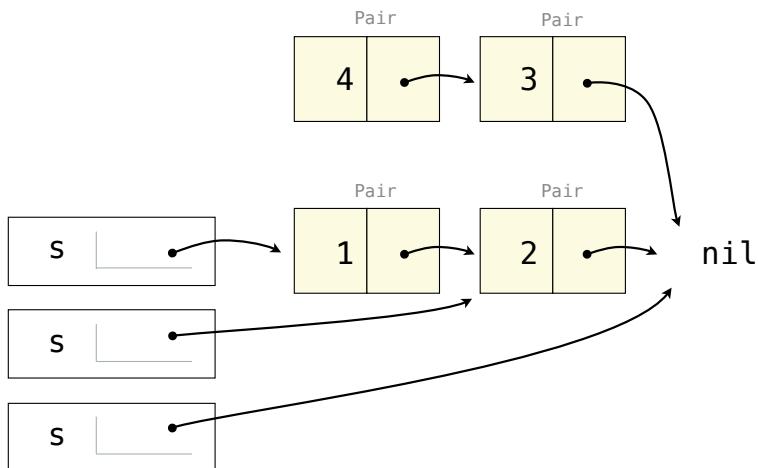
(reduce (lambda (x y) (cons y x)) '(3 4 5) '(2))

(5 4 3 2)

Example: Map with Only a Constant Number of Frames

```
(define (map procedure s)
  (if (null? s)
      nil
      (cons (procedure (car s))
            (map procedure (cdr s))))) )
```

```
(map (lambda (x) (- 5 x)) (list 1 2))
```



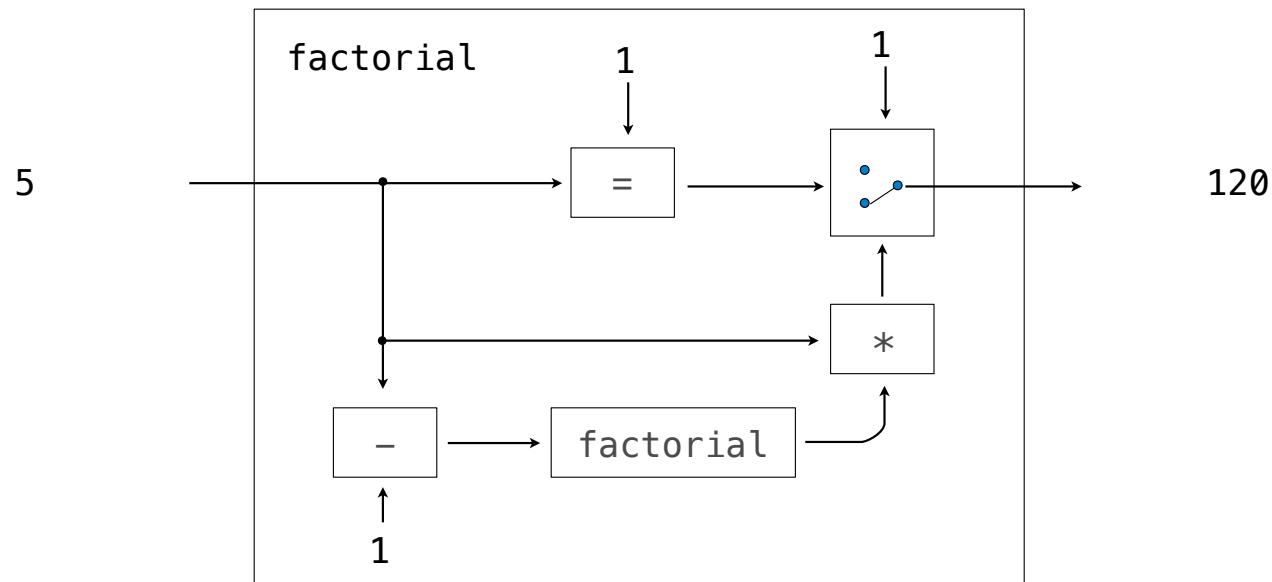
```
(define (map procedure s)
  (define (map-reverse s m)
    (if (null? s)
        m
        (map-reverse (cdr s)
                     (cons (procedure (car s))
                           m)))) )
  (reverse (map-reverse s nil)))
```

```
(define (reverse s)
  (define (reverse-iter s r)
    (if (null? s)
        r
        (reverse-iter (cdr s)
                     (cons (car s) r)) ) )
  (reverse-iter s nil)) )
```

General Computing Machines

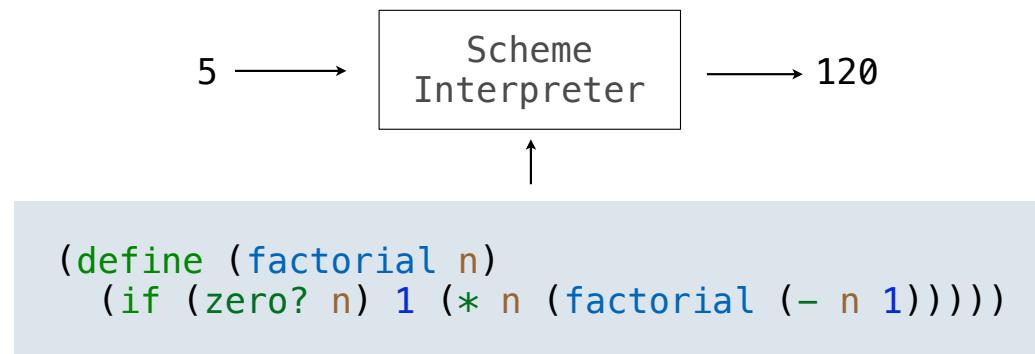
An Analogy: Programs Define Machines

Programs specify the logic of a computational device



Interpreters are General Computing Machine

An interpreter can be parameterized to simulate any machine



Our Scheme interpreter is a universal machine

A bridge between the data objects that are manipulated by our programming language and the programming language itself

Internally, it is just a set of evaluation rules

Programs as Data

A Scheme Expression is a Scheme List

Scheme programs consist of expressions, which can be:

- Primitive expressions: 2 3.3 true + quotient
- Combinations: (quotient 10 2) (not true)

The built-in Scheme list data structure (which is a linked list) can represent combinations

```
scm> (list 'quotient 10 2)
(quotient 10 2)
```

```
scm> (eval (list 'quotient 10 2))
5
```

In such a language, it is straightforward to write a program that writes a program

(Demo)

Macros

Macros Perform Code Transformations

A macro is an operation performed on the source code of a program before evaluation

Macros exist in many languages, but are easiest to define correctly in a language like Lisp

Scheme has a **define-macro** special form that defines a source code transformation

```
(define-macro (twice expr)
  (list 'begin expr expr))
> (twice (print 2)) ➤ (begin (print 2) (print 2))
2
2
```

Evaluation procedure of a macro call expression:

- Evaluate the operator sub-expression, which evaluates to a macro
- Call the macro procedure on the operand expressions without evaluating them first
- Evaluate the expression returned from the macro procedure

(Demo)

For Macro

Discussion Question

Define a macro that evaluates an expression for each value in a sequence

```
(define (map fn vals)
  (if (null? vals)
      ()
      (cons (fn (car vals))
            (map fn (cdr vals)))))

scm> (map (lambda (x) (* x x)) '(2 3 4 5))
(4 9 16 25)

(define-macro (for sym vals expr)
  (list 'map _____
        (list 'lambda (list sym) expr) vals))

scm> (for x '(2 3 4 5) (* x x))
(4 9 16 25)
```

(Demo)

Quasi-Quotation

(Demo)

Streams

Announcements

Efficient Sequence Processing

Sequence Operations

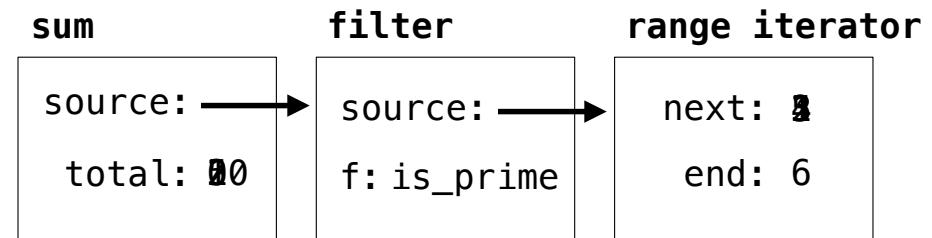
Map, filter, and reduce express sequence manipulation using compact expressions

Example: Sum all primes in an interval from **a** (inclusive) to **b** (exclusive)

```
def sum_primes(a, b):
    total = 0
    x = a
    while x < b:
        if is_prime(x):
            total = total + x
        x = x + 1
    return total
```

```
def sum_primes(a, b):
    return sum(filter(is_prime, range(a, b)))
```

sum_primes(1, 6)



Space:

Constant

Also Constant

(Demo)

Streams

Streams are Lazy Scheme Lists

A stream is a list, but the rest of the list is computed only when needed:

(car (cons 1 nil)) \rightarrow 1	(car (cons-stream 1 nil)) \rightarrow 1
(cdr (cons 1 nil)) \rightarrow ()	(cdr-stream (cons-stream 1 nil)) \rightarrow ()
(cons 1 (cons 2 nil))	(cons-stream 1 (cons-stream 2 nil))

Errors only occur when expressions are evaluated:

(cons 1 (cons (/ 1 0) nil))	\rightarrow ERROR
(cons-stream 1 (cons-stream (/ 1 0) nil))	\rightarrow (1 . #[promise (not forced)])
(car (cons-stream 1 (cons-stream (/ 1 0) nil)))	\rightarrow 1
(cdr-stream (cons-stream 1 (cons-stream (/ 1 0) nil)))	\rightarrow ERROR

(Demo)

Stream Ranges are Implicit

A stream can give on-demand access to each element in order

```
(define (range-stream a b)
  (if (>= a b)
      nil
      (cons-stream a (range-stream (+ a 1) b)))))

(define lots (range-stream 1 1000000000000000000))

scm> (car lots)
1
scm> (car (cdr-stream lots))
2
scm> (car (cdr-stream (cdr-stream lots)))
3
```

Infinite Streams

Integer Stream

An integer stream is a stream of consecutive integers

The rest of the stream is not yet computed when the stream is created

```
(define (int-stream start)
  (cons-stream start (int-stream (+ start 1)))))
```

(Demo)

Stream Processing

(Demo)

Recursively Defined Streams

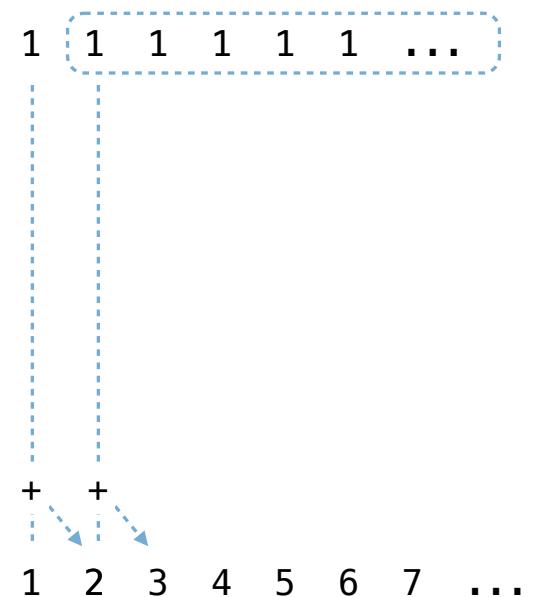
The rest of a constant stream is the constant stream

```
(define ones (cons-stream 1 ones))
```

Combine two streams by separating each into car and cdr

```
(define (add-streams s t)
  (cons-stream (+ (car s) (car t))
               (add-streams (cdr-stream s)
                           (cdr-stream t))))
```

```
(define ints (cons-stream 1 (add-streams ones ints)))
```



Higher-Order Stream Functions

Higher-Order Functions on Streams

Implementations are identical,
but change cons to cons-stream
and change cdr to cdr-stream

```
(define (map-stream f s)
  (if (null? s)
      nil
      (cons-stream (f (car s))
                   (map-stream f
                               (cdr-stream s)))))

(define (filter-stream f s)
  (if (null? s)
      nil
      (if (f (car s))
          (cons-stream (car s)
                       (filter-stream f (cdr-stream s)))
          (filter-stream f (cdr-stream s)))))

(define (reduce-stream f t start)
  (if (null? s)
      start
      (reduce-stream f
                     (cdr-stream s)
                     (f start (car s)))))
```

:%s/\v(map|filter|reduce|cdr|cons)/\1-stream/g

14

A Stream of Primes

For any prime k , any larger prime must not be divisible by k .

The stream of integers not divisible by any $k \leq n$ is:

The stream of integers not divisible by any $k < n$

Filtered to remove any element divisible by n

This recurrence is called the Sieve of Eratosthenes

2, 3, ~~4~~, ~~5~~, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13



(Demo)

Declarative Languages

Database Management Systems

Database management systems (DBMS) are important, heavily used, and interesting!

A table is a collection of records, which are rows that have a value for each column

Latitude	Longitude	Name
38	122	Berkeley
42	71	Cambridge
45	93	Minneapolis

A table has columns and rows

A row has a value for each column

A column has a name and a type

The Structured Query Language (SQL) is perhaps the most widely used programming language

SQL is a *declarative* programming language

Declarative Programming

In **declarative languages** such as SQL & Prolog:

- A "program" is a description of the desired result
- The interpreter figures out how to generate the result

In **imperative languages** such as Python & Scheme:

- A "program" is a description of computational processes
- The interpreter carries out execution/evaluation rules

```
create table cities as
  select 38 as latitude, 122 as longitude, "Berkeley" as name union
  select 42,           71,           "Cambridge"
  select 45,           93,           "Minneapolis";
```

```
select "west coast" as region, name from cities where longitude >= 115 union
select "other",           name from cities where longitude <  115;
```

Cities:

latitude	longitude	name
38	122	Berkeley
42	71	Cambridge
45	93	Minneapolis

region	name
west coast	Berkeley
other	Minneapolis
other	Cambridge

Structured Query Language (SQL)

SQL Overview

The SQL language is an ANSI and ISO standard, but DBMS's implement custom variants

- A **select** statement creates a new table, either from scratch or by projecting a table
- A **create table** statement gives a global name to a table
- Lots of other statements exist: **analyze**, **delete**, **explain**, **insert**, **replace**, **update**, etc.
- Most of the important action is in the **select** statement

Today's theme:



<http://awhimsicalbohemian.typepad.com/.a/6a00e5538b84f3883301538dfa8f19970b-800wi>

7

Getting Started with SQL

Install sqlite (version 3.8.3 or later): <http://sqlite.org/download.html>

Use sqlite online: code.cs61a.org/sql

Selecting Value Literals

A **select** statement always includes a comma-separated list of column descriptions

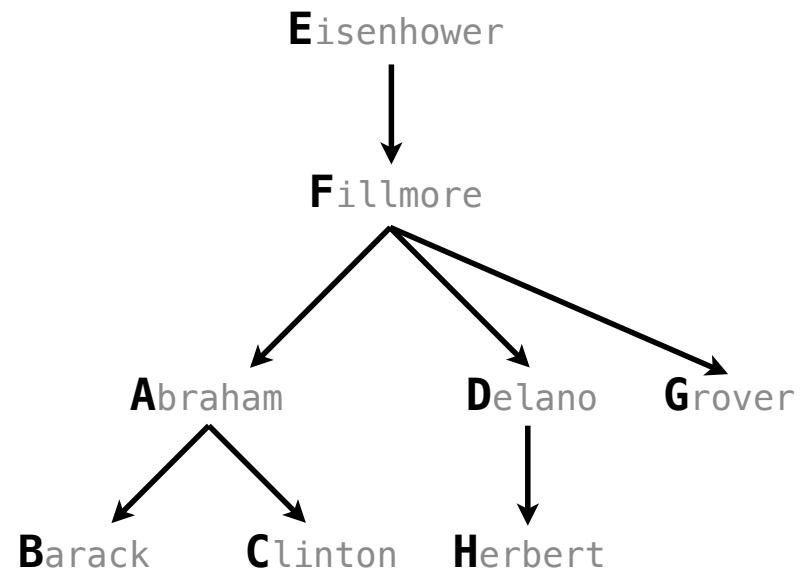
A column description is an expression, optionally followed by **as** and a column name

```
select [expression] as [name], [expression] as [name]; ...
```

Selecting literals creates a one-row table

The union of two select statements is a table containing the rows of both of their results

```
select "delano" as parent, "herbert" as child;union
select "abraham"      , "barack"           union
select "abraham"      , "clinton"          union
select "fillmore"     , "abraham"          union
select "fillmore"     , "delano"           union
select "fillmore"     , "grover"           union
select "eisenhower"   , "fillmore";
```



Naming Tables

SQL is often used as an interactive language

The result of a **select** statement is displayed to the user, but not stored

A **create table** statement gives the result a name

```
create table [name] as [select statement];
```

```
create table parents as
select "delano" as parent, "herbert" as child union
select "abraham"      , "barack"           union
select "abraham"      , "clinton"          union
select "fillmore"     , "abraham"          union
select "fillmore"     , "delano"           union
select "fillmore"     , "grover"           union
select "eisenhower"   , "fillmore";
```

Parents:

Parent	Child
abraham	barack
abraham	clinton
delano	herbert
fillmore	abraham
fillmore	delano
fillmore	grover
eisenhower	fillmore

Projecting Tables

Select Statements Project Existing Tables

A **select** statement can specify an input table using a **from** clause

A subset of the rows of the input table can be selected using a **where** clause

An ordering over the remaining rows can be declared using an **order by** clause

Column descriptions determine how each input row is projected to a result row

```
select [expression] as [name], [expression] as [name], ...;
```

```
select [columns] from [table] where [condition] order by [order];
```

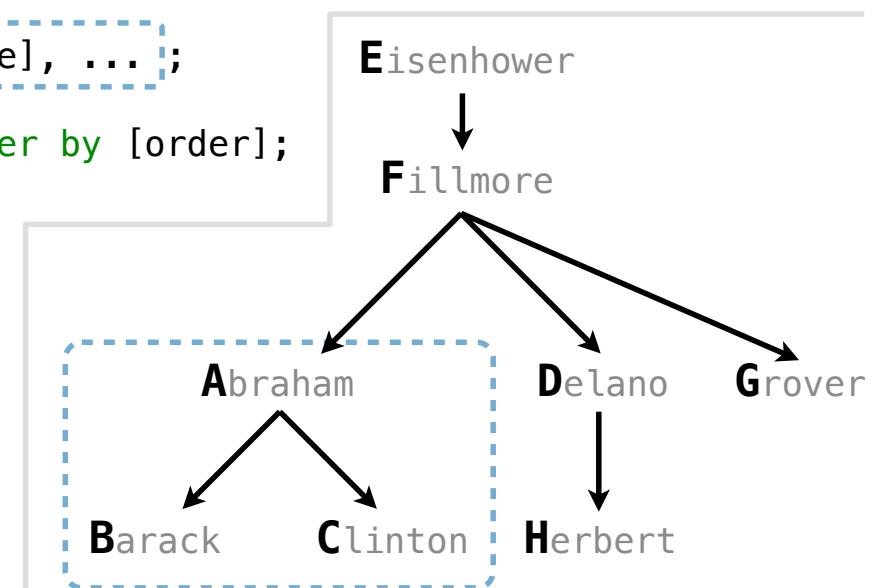
```
select child from parents where parent = "abraham";
```

```
select parent from parents where parent > child;
```

Child
barack
clinton

Parent
fillmore
fillmore

(Demo)



Arithmetic

Arithmetic in Select Expressions

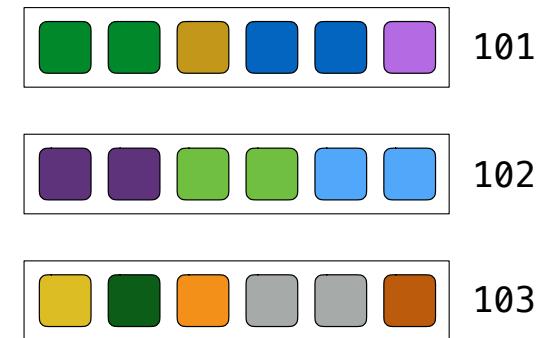
In a select expression, column names evaluate to row values

Arithmetic expressions can combine row values and constants

```
create table lift as
  select 101 as chair, 2 as single, 2 as couple union
  select 102      , 0           , 3             union
  select 103      , 4           , 1;
```

```
select chair, single + 2 * couple as total from lift;
```

chair	total
101	6
102	6
103	6



Discussion Question

Given the table **ints** that describes how to sum powers of 2 to form various integers

```
create table ints as
    select "zero" as word, 0 as one, 0 as two, 0 as four, 0 as eight union
    select "one"      , 1      , 0      , 0      , 0      union
    select "two"      , 0      , 2      , 0      , 0      union
    select "three"    , 1      , 2      , 0      , 0      union
    select "four"     , 0      , 0      , 4      , 0      union
    select "five"     , 1      , 0      , 4      , 0      union
    select "six"      , 0      , 2      , 4      , 0      union
    select "seven"    , 1      , 2      , 4      , 0      union
    select "eight"    , 0      , 0      , 0      , 8      union
    select "nine"     , 1      , 0      , 0      , 8;
```

(A) Write a select statement for a two-column table of the **word** and **value** for each integer

word	value
zero	0
one	1
two	2
three	3
...	...

(Demo)

(B) Write a select statement for the **word** names of the powers of two

word
one
two
four
eight

Joining Tables

Reminder: John the Patriotic Dog Breeder



```
CREATE TABLE parents AS  
  
SELECT "abraham" AS parent, "barack" AS child UNION  
SELECT "abraham" , "clinton" UNION  
SELECT "delano" , "herbert" UNION  
SELECT "fillmore" , "abraham" UNION  
SELECT "fillmore" , "delano" UNION  
SELECT "fillmore" , "grover" UNION  
SELECT "eisenhower" , "fillmore";
```

Parents:

Parent	Child
abraham	barack
abraham	clinton
delano	herbert
fillmore	abraham
fillmore	delano
fillmore	grover
eisenhower	fillmore

Joining Two Tables

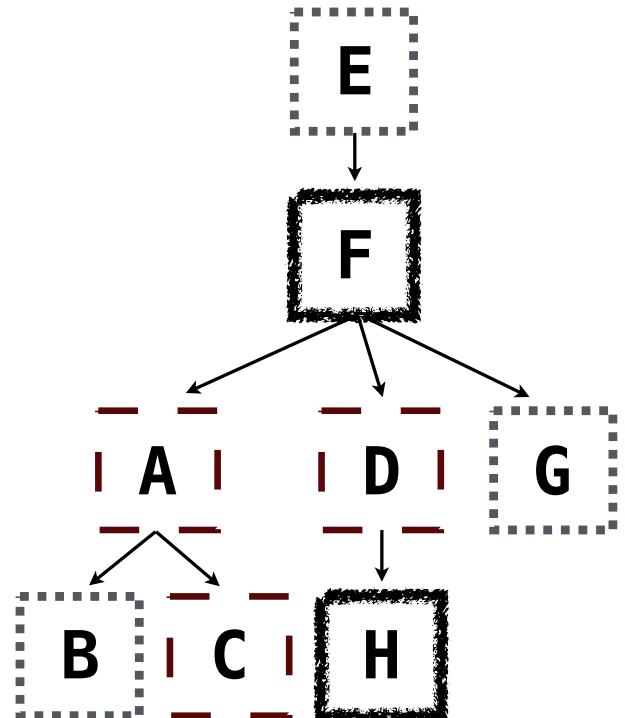
Two tables **A** & **B** are joined by a comma to yield all combos of a row from **A** & a row from **B**

```
CREATE TABLE dogs AS
SELECT "abraham" AS name, "long" AS fur UNION
SELECT "barack" , "short" UNION
SELECT "clinton" , "long" UNION
SELECT "delano" , "long" UNION
SELECT "eisenhower" , "short" UNION
SELECT "fillmore" , "curly" UNION
SELECT "grover" , "short" UNION
SELECT "herbert" , "curly";
```

```
CREATE TABLE parents AS
SELECT "abraham" AS parent, "barack" AS child UNION
SELECT "abraham" , "clinton" UNION
...;
```

Select the parents of curly-furred dogs

```
SELECT parent FROM parents, dogs
WHERE child = name AND fur = "curly";
```



(Demo)

Aliases and Dot Expressions

Joining a Table with Itself

Two tables may share a column name; dot expressions and aliases disambiguate column values

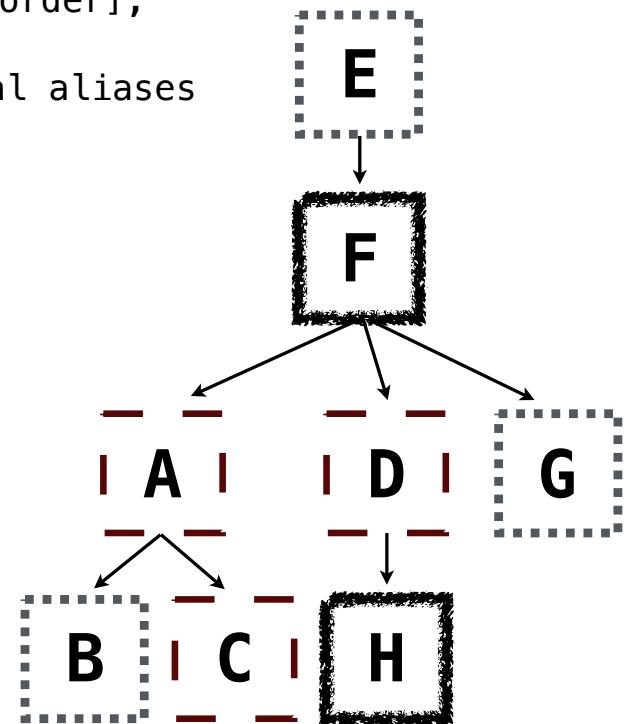
```
SELECT [columns] FROM [table] WHERE [condition] ORDER BY [order];
```

[table] is a comma-separated list of table names with optional aliases

Select all pairs of siblings

```
SELECT a.child AS first, b.child AS second  
FROM parents AS a, parents AS b  
WHERE a.parent = b.parent AND a.child < b.child;
```

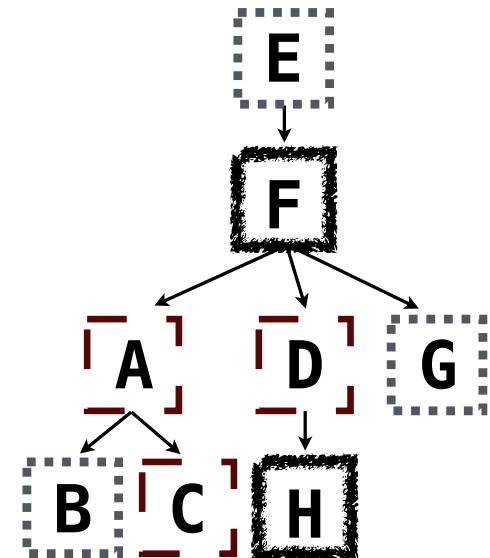
First	Second
barack	clinton
abraham	delano
abraham	grover
delano	grover



Example: Grandparents

Which select statement evaluates to all grandparent, grandchild pairs?

- 1 `SELECT a.grandparent, b.child FROM parents AS a, parents AS b WHERE b.parent = a.child;`
- 2 `SELECT a.parent, b.child FROM parents AS a, parents AS b WHERE a.parent = b.child;`
- 3 `SELECT a.parent, b.child FROM parents AS a, parents AS b WHERE b.parent = a.child;`
- 4 `SELECT a.grandparent, b.child FROM parents AS a, parents AS b WHERE a.parent = b.parent;`
- 5 None of the above



Joining Multiple Tables

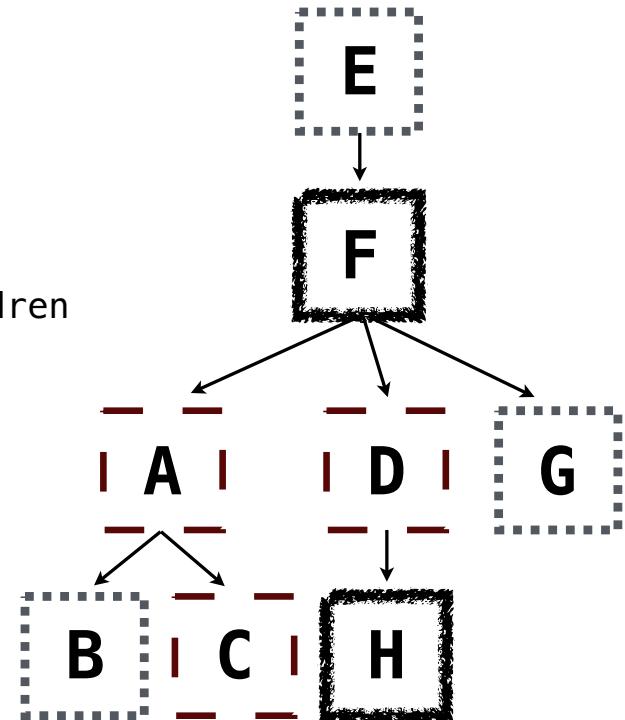
Multiple tables can be joined to yield all combinations of rows from each

```
CREATE TABLE grandparents AS  
  SELECT a.parent AS grandog, b.child AS granup  
    FROM parents AS a, parents AS b  
   WHERE b.parent = a.child;
```

Select all grandparents with the same fur as their grandchildren

Which tables need to be joined together?

```
SELECT grandog FROM grandparents, dogs AS c, dogs AS d  
  WHERE grandog = c.name AND  
        granup = d.name AND  
        c.fur = d.fur;
```



Example: Dog Triples

Fall 2014 Quiz Question (Slightly Modified)

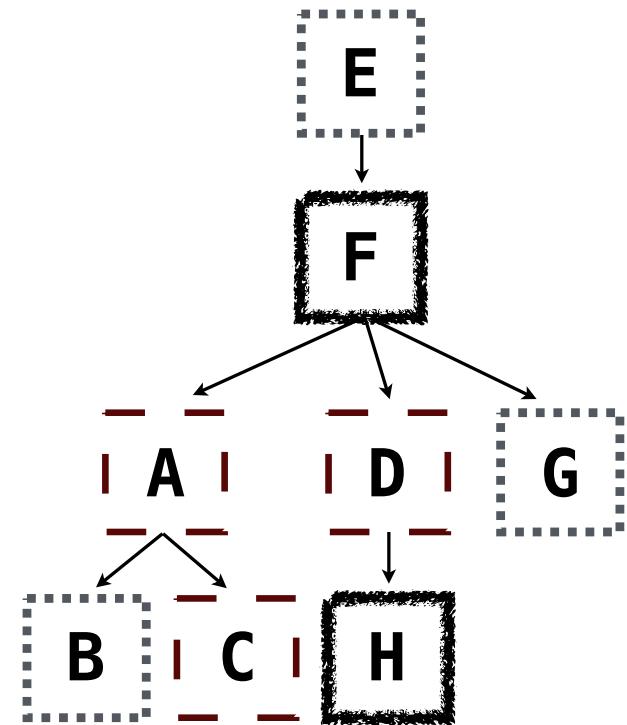
Write a SQL query that selects all possible combinations of three different dogs with the same fur and lists each triple in *inverse alphabetical order*

```
CREATE TABLE dogs AS
  SELECT "abraham" AS name, "long" AS fur UNION
  SELECT "barack"      , "short"    UNION
  ...
CREATE TABLE parents AS
  SELECT "abraham" AS parent, "barack" AS child UNION
  SELECT "abraham"      , "clinton"  UNION
  ...;
```

Expected output:

delano|clinton|abraham
grover|eisenhower|barack

(Demo)



Numerical Expressions

Numerical Expressions

Expressions can contain function calls and arithmetic operators



```
SELECT [columns] FROM [table] WHERE [expression] ORDER BY [expression];
```

Combine values: +, -, *, /, %, and, or

Transform values: abs, round, not, -

Compare values: <, <=, >, >=, <>, !=, =

(Demo)

String Expressions

String Expressions

String values can be combined to form longer strings



```
sqlite> SELECT "hello," || " world";
hello, world
```

Basic string manipulation is built into SQL, but differs from Python



```
sqlite> CREATE TABLE phrase AS SELECT "hello, world" AS s;
sqlite> SELECT substr(s, 4, 2) || substr(s, instr(s, " ")+1, 1) FROM phrase;
low
```

Strings can be used to represent structured values, but doing so is rarely a good idea



```
sqlite> CREATE TABLE lists AS SELECT "one" AS car, "two,three,four" AS cdr;
sqlite> SELECT substr(cdr, 1, instr(cdr, ",")-1) AS caddr FROM lists;
two
```

(Demo)

Aggregation

Aggregate Functions

So far, all SQL expressions have referred to the values in a single row at a time

```
[expression] as [name], [expression] as [name], ...  
select [columns] from [table] where [expression] order by [expression];
```

An aggregate function in the [columns] clause computes a value from a group of rows

```
create table animals as  
select "dog" as kind, 4 as legs, 20 as weight union  
select "cat" , 4 , 10 union  
select "ferret" , 4 , 10 union  
select "parrot" , 2 , 6 union  
select "penguin" , 2 , 10 union  
select "t-rex" , 2 , 12000;  
  
select max(legs) from animals;
```

max(legs)
4

(Demo)

animals:

kind	legs	weight
dog	4	20
cat	4	10
ferret	4	10
parrot	2	6
penguin	2	10
t-rex	2	12000

Mixing Aggregate Functions and Single Values

An aggregate function also selects some row in the table to supply the values of columns that are not aggregated. In the case of max or min, this row is that of the max or min value. Otherwise, it is arbitrary.

```
select max(weight), kind from animals;
```

```
select min(kind), kind from animals;
```

```
select max(legs), kind from animals;
```

```
select avg(weight), kind from animals;
```

(Demo)

```
create table animals as
select "dog" as kind, 4 as legs, 20 as weight union
select "cat"      , 4      , 10      union
select "ferret"   , 4      , 10      union
select "parrot"   , 2      , 6       union
select "penguin"  , 2      , 10      union
select "t-rex"    , 2      , 12000;
```

animals:

kind	legs	weight
dog	4	20
cat	4	10
ferret	4	10
parrot	2	6
penguin	2	10
t-rex	2	12000

Discussion Question

What are all the kinds of animals that have the maximal number of legs?

Groups

Grouping Rows

Rows in a table can be grouped, and aggregation is performed on each group

```
[expression] as [name], [expression] as [name], ...  
select [columns] from [table] group by [expression] having [expression];
```

The number of groups is the number of unique values of an expression

```
select legs, max(weight) from animals group by legs;
```

legs	max(weight)
4	20
2	12000

legs=4
legs=2
(Demo)

animals:

kind	legs	weight
dog	4	20
cat	4	10
ferret	4	10
parrot	2	6
penguin	2	10
t-rex	2	12000

Selecting Groups

Rows in a table can be grouped, and aggregation is performed on each group

```
[expression] as [name], [expression] as [name], ...  
select [columns] from [table] group by [expression] having [expression];
```

A `having` clause filters the set of groups that are aggregated

```
select weight/legs, count(*) from animals group by weight/legs having count(*)>1;
```

weight/legs	count(*)
5	2
2	2

weight/legs=5
weight/legs=2
weight/legs=2
weight/legs=3
weight/legs=5
weight/legs=6000

animals:

kind	legs	weight
dog	4	20
cat	4	10
ferret	4	10
parrot	2	6
penguin	2	10
t-rex	2	12000

Discussion Question

What's the maximum difference between leg count for two animals with the same weight?

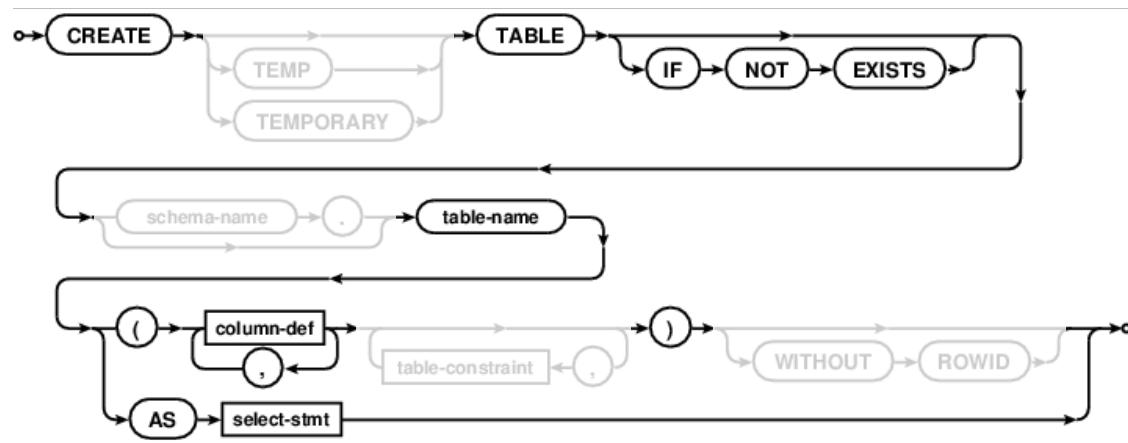
Example: Big Game

(Demo)

Create Table and Drop Table

Create Table

CREATE TABLE expression syntax:



Examples:

`CREATE TABLE numbers (n, note);`

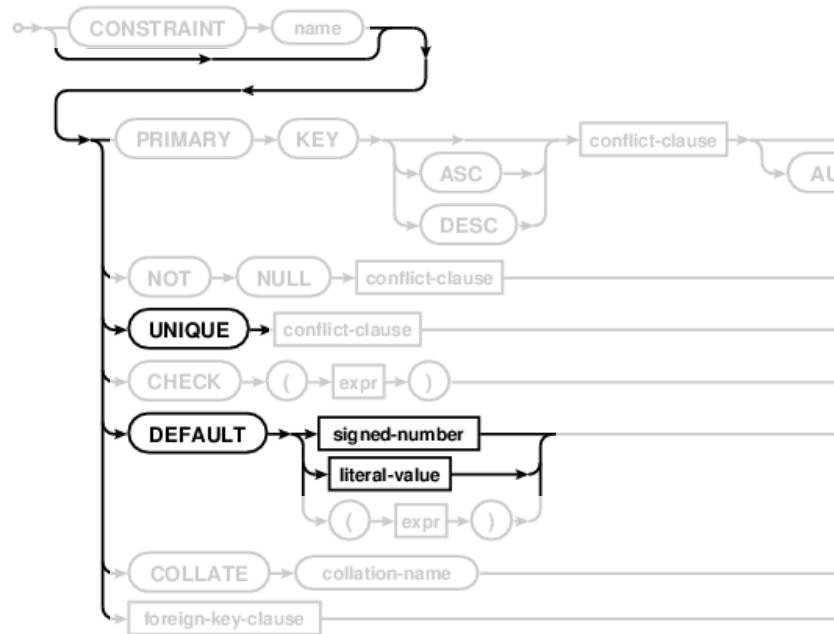
`CREATE TABLE numbers (n UNIQUE, note);`

`CREATE TABLE numbers (n, note DEFAULT "No comment");`

column-def:



column-constraint:

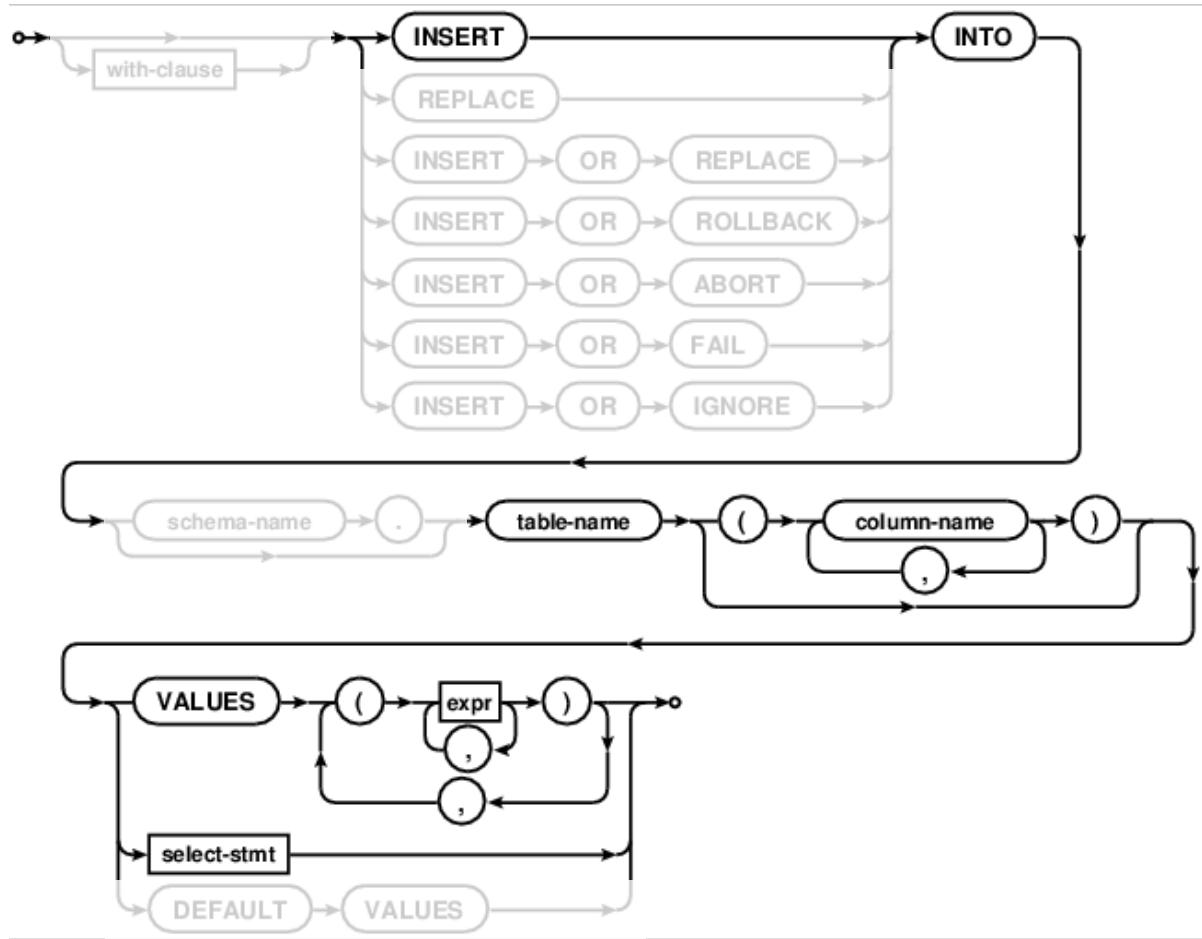


Drop Table



Modifying Tables

Insert



For a table t with two columns...

To insert into one column:

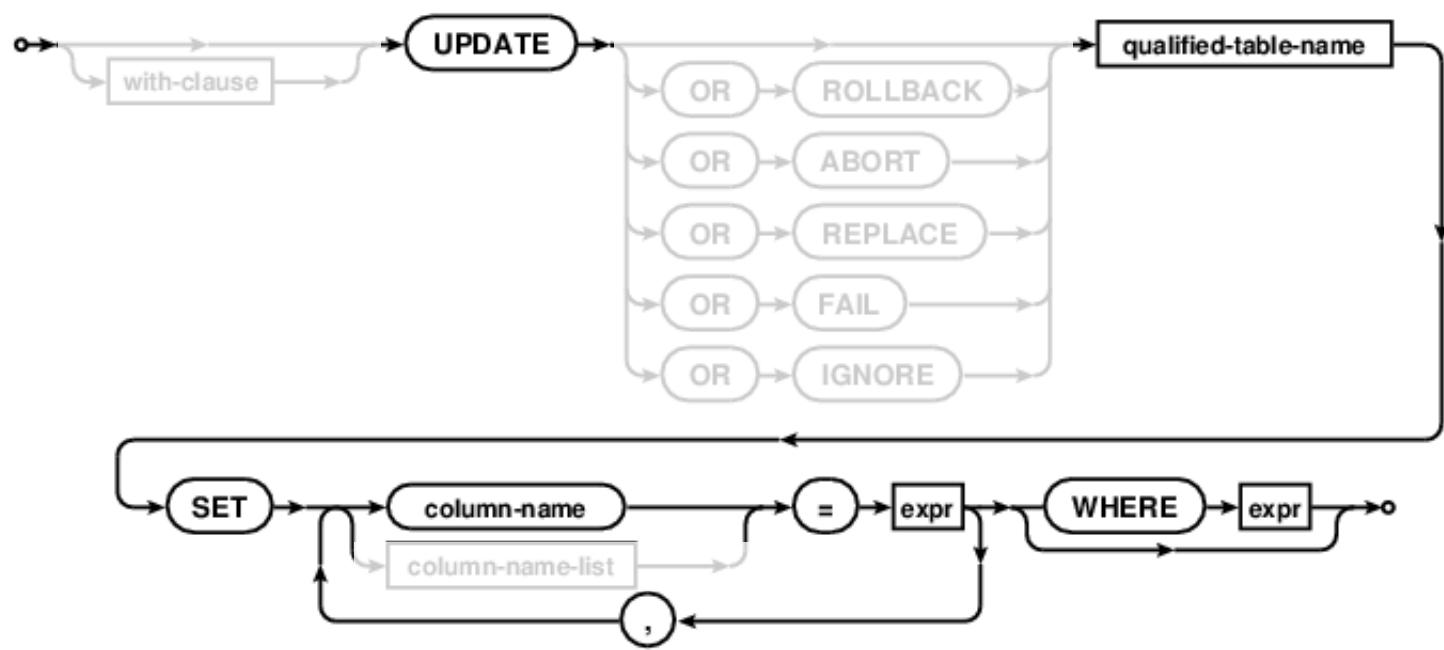
`INSERT INTO t(column) VALUES (value);`

To insert into both columns:

`INSERT INTO t VALUES (value0, value1);`

(Demo)

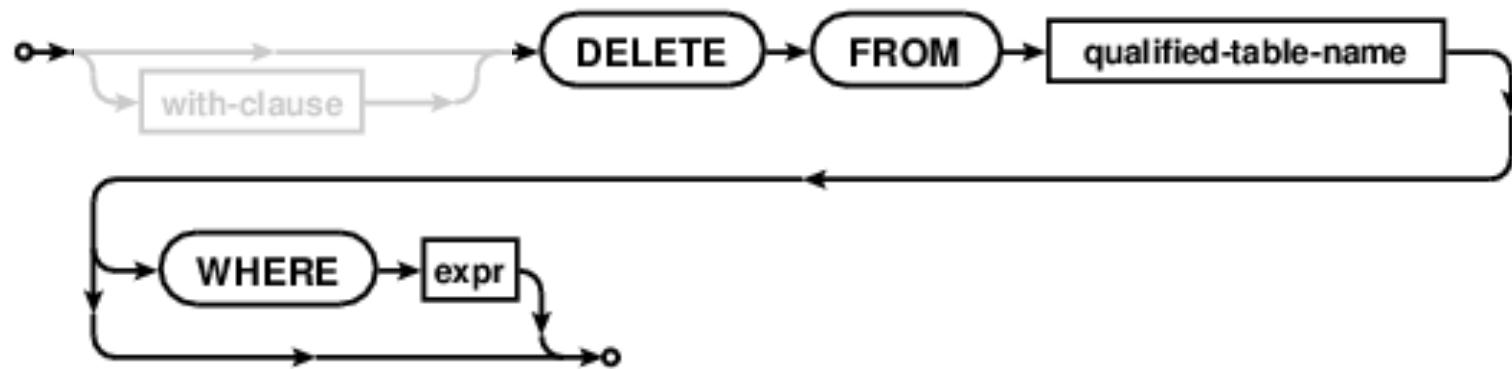
Update



Update sets all entries in certain columns to new values, just for some subset of rows.

(Demo)

Delete



Delete removes some or all rows from a table.

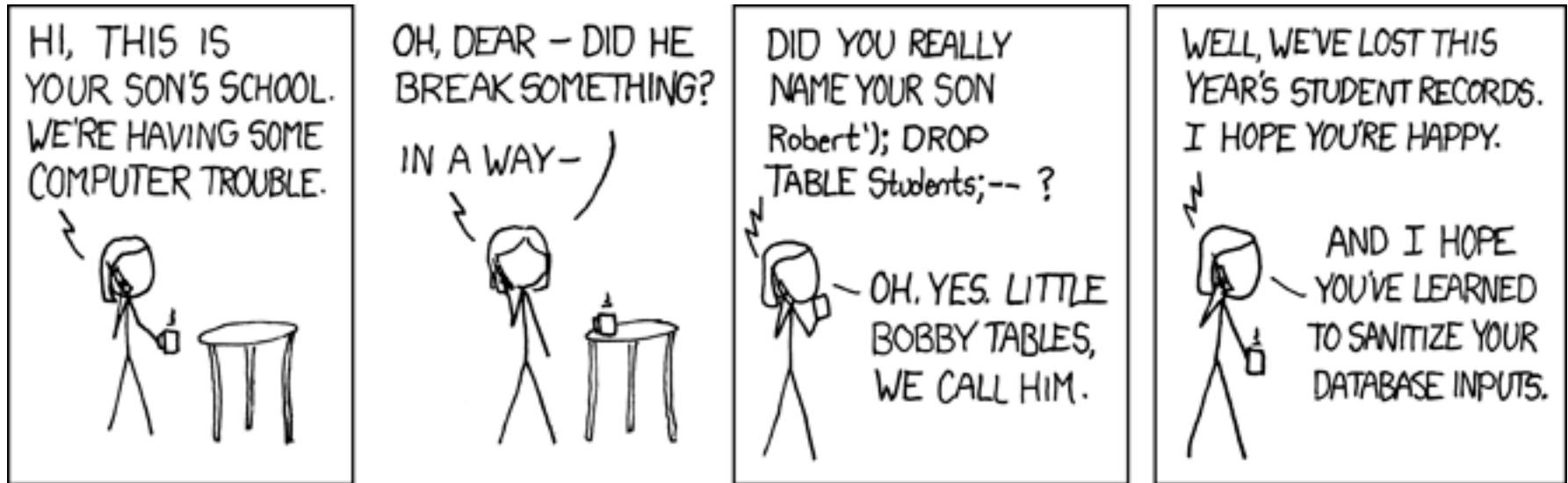
(Demo)

Python and SQL

(Demo)

SQL Injection Attack

A Program Vulnerable to a SQL Injection Attack



```
name = "Robert'); DROP TABLE Students; --"  
cmd = "INSERT INTO Students VALUES ('" + name + "');"  
db.executescript(cmd) db.execute("INSERT INTO Students VALUES (?)", [name])  
  
INSERT INTO Students VALUES ('Robert'''); DROP TABLE Students; --';  
INSERT INTO Students VALUES ('Robert'); DROP TABLE Students; '');
```

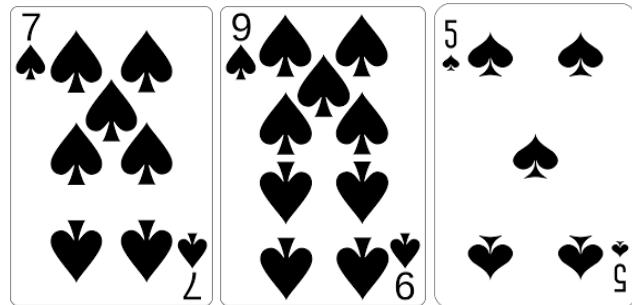
<https://xkcd.com/327/>

12

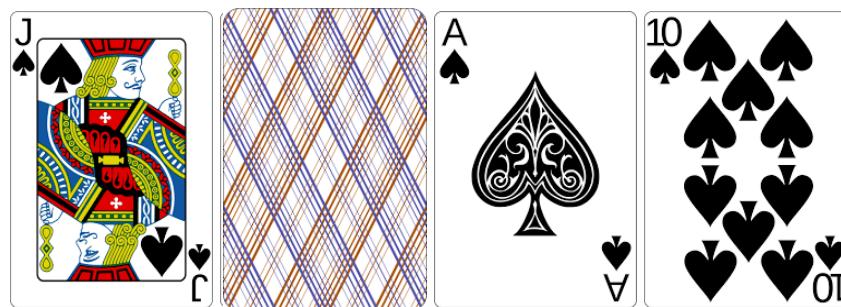
Database Connections

Casino Blackjack

Player:



Dealer:



(Demo)

Trees

Tree-Structured Data

```
def tree(label, branches=[]):
    return [label] + list(branches)

def label(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        self.branches = list(branches)
```

A tree can contains other trees:

[5, [6, 7], 8, [[9], 10]]

(+ 5 (- 6 7) 8 (* (- 9) 10))

(S
 (NP (JJ Short) (NNS cuts))
 (VP (VBP make)
 (NP (JJ long) (NNS delays)))
 (. .))

 Midterm 1
 Midterm 2

Tree processing often involves recursive calls on subtrees

Tree Processing

Solving Tree Problems

Implement `bigs`, which takes a Tree instance `t` containing integer labels. It returns the number of nodes in `t` whose labels are larger than any labels of their ancestor nodes.

```
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors.
```

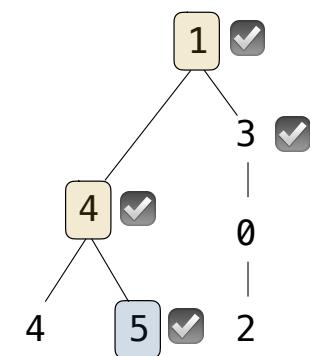
```
>>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
>>> bigs(a)
4
"""
```

The root label is always larger than all of its ancestors

```
if t.is_leaf():
    return __
else:
    return __([__ for b in t.branches])
```

Somehow increment
the total count

```
Somehow track a
list of ancestors
if node.label > max(ancestors):
    Somehow track the
    largest ancestor
if node.label > max_ancestors:
```

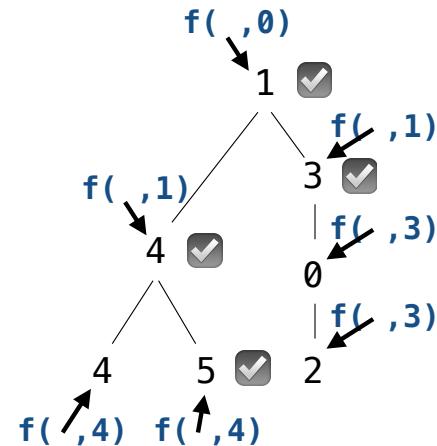


Solving Tree Problems

Implement `bigs`, which takes a Tree instance `t` containing integer labels. It returns the number of nodes in `t` whose labels are larger than any labels of their ancestor nodes.

```
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors.

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """
    def f(a, x):
        """Somehow track the largest ancestor
        A node in t
        if max_ancestor
            if a.label > x
                return 1 + sum([f(b, a.label) for b in a.branches])
            else:
                somehow increment the total count
                return sum([f(b, x) for b in a.branches])
        return f(t, t.label - 1)
    Some initial value for the largest ancestor so far...
```



Recursive Accumulation

Solving Tree Problems

Implement **bigs**, which takes a Tree instance *t* containing integer labels. It returns the number of nodes in *t* whose labels are larger than any labels of their ancestor nodes.

```
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors."""
    n = 0
    def f(a, x):
        Somehow track the
        largest ancestor
        nonlocal n
        if a.label > x:
            node.label > max_ancestors
            n += 1
            Somehow increment
            the total count
        for b in a.branches:
            f(b, max(a.label, x))
    f(t, t.label - 1)
    return n
```

Somehow track the largest ancestor

node.label > max_ancestors

Somehow increment the total count

Root label is always larger than its ancestors

Designing Functions

How to Design Programs

From Problem Analysis to Data Definitions

Identify the information that must be represented and how it is represented in the chosen programming language. Formulate data definitions and illustrate them with examples.

Signature, Purpose Statement, Header

State what kind of data the desired function consumes and produces. Formulate a concise answer to the question *what* the function computes. Define a stub that lives up to the signature.

Functional Examples

Work through examples that illustrate the function's purpose.

Function Template

Translate the data definitions into an outline of the function.

Function Definition

Fill in the gaps in the function template. Exploit the purpose statement and the examples.

Testing

Articulate the examples as tests and ensure that the function passes all. Doing so discovers mistakes. Tests also supplement examples in that they help others read and understand the definition when the need arises—and it will arise for any serious program.

Applying the Design Process

Designing a Function

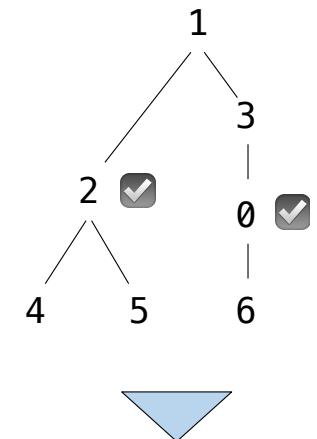
Implement `smalls`, which takes a Tree instance `t` containing integer labels. It returns the non-leaf nodes in `t` whose labels are smaller than any labels of their descendant nodes.

```
def smalls(t):      Signature: Tree -> List of Trees
    """Return the non-leaf nodes in t that are smaller than all their descendants.

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)]])
    [0, 2]

    """
    result = []          Signature: Tree -> number
    def process(t):     "Find smallest label in t & maybe add t to result"
        if t.is_leaf():
            return t.label
        else:
            return min(...)

    process(t)
    return result
```



```
[ 2 , 0 ]
```

Designing a Function

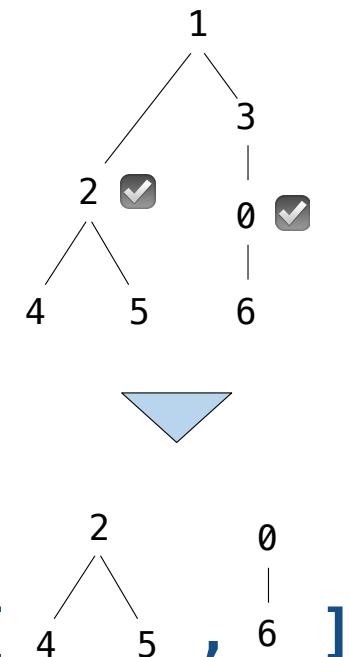
Implement `smalls`, which takes a Tree instance `t` containing integer labels. It returns the non-leaf nodes in `t` whose labels are smaller than any labels of their descendant nodes.

```
def smalls(t):      Signature: Tree -> List of Trees
    """Return the non-leaf nodes in t that are smaller than all their descendants.

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]

    """
    result = []          Signature: Tree -> number
    def process(t):     "Find smallest label in t & maybe add t to result"
        if t.is_leaf():
            return _____ t.label
        else:
            smallest = _____ min([process(b) for b in t.branches])
            _____
            if _____ t.label < smallest _____:
                result.append( t )
            _____
            return min(smallest, t.label)
    process(t)
    return result
```

*smallest label
in a branch of t* → *if _____ t.label < smallest _____:*



Expression Trees

Interpreter Analysis

How many times does scheme_eval get called when evaluating the following expressions?

(define x (+ 1 2))

(define (f y) (+ x y))

(f (if (> 3 2) 4 5))