UC Berkeley EECS
Adj. Ass. Prof.
Dr. Gerald Friedland

Computational Structures in Data Science

Lecture #13: Review

April 29th, 2019

http://inst.eecs.berkeley.edu/~cs88

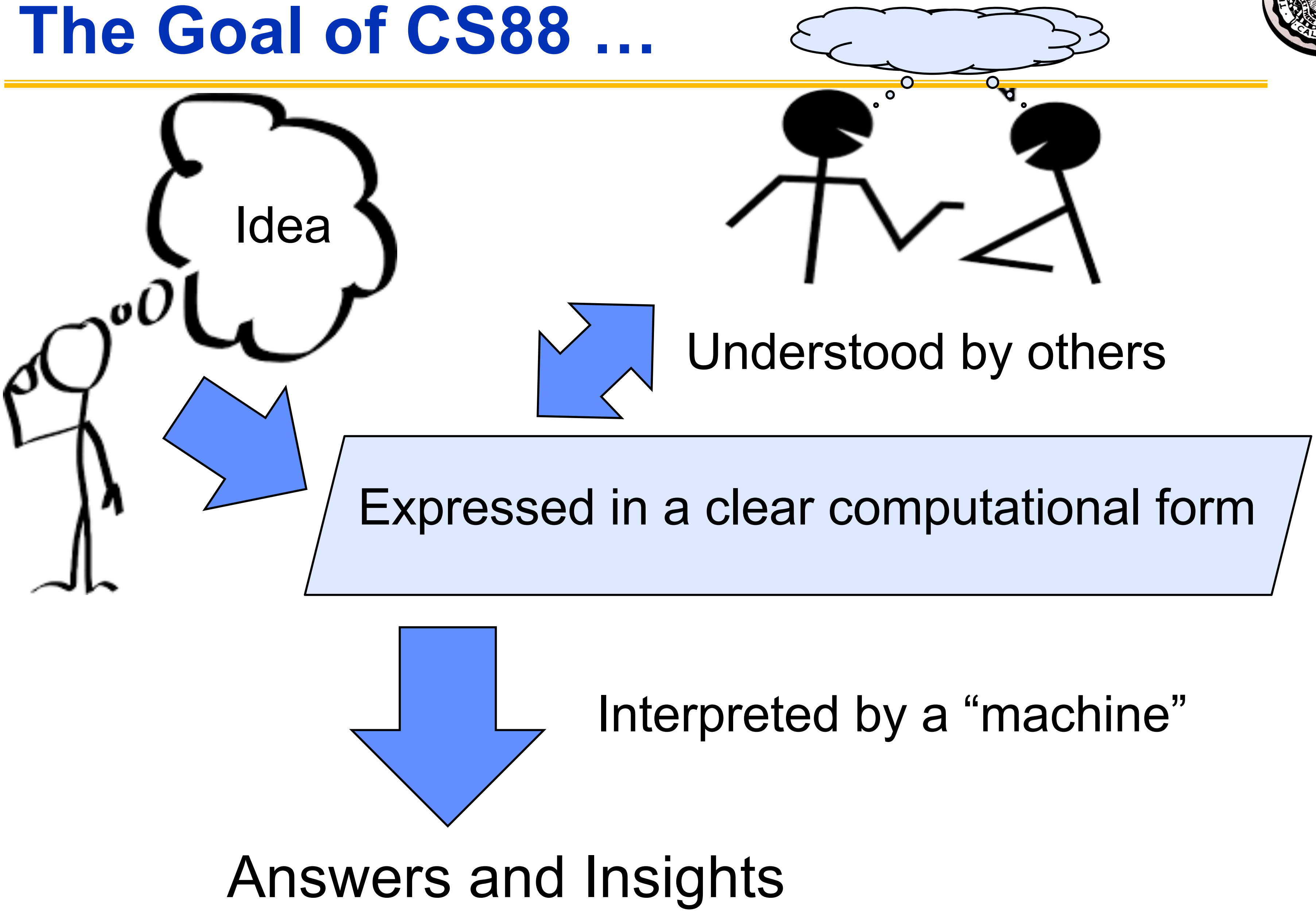
Computational Concepts Toolbox

- Data type: values, literals, operations,
- Expressions, Call expression
- Variables
- Assignment Statement, Tuple assignment
- Sequences: tuple, list
- Dictionaries
- Function Definition Statement
- Conditional Statement
- Iteration: list comp, for, while
- Lambda function expr.

SQL

- Higher Order Functions
  - Functions as Values
  - Functions with functions as argument
  - Assignment of function values
- Higher order function patterns
  - Map, Filter, Reduce
- Function factories – create and return functions
- Recursion
- Abstract Data Types
- Mutation
- Class & Inheritance
- Exceptions
- Iterators & Generators

The Goal of CS88 …



Idea

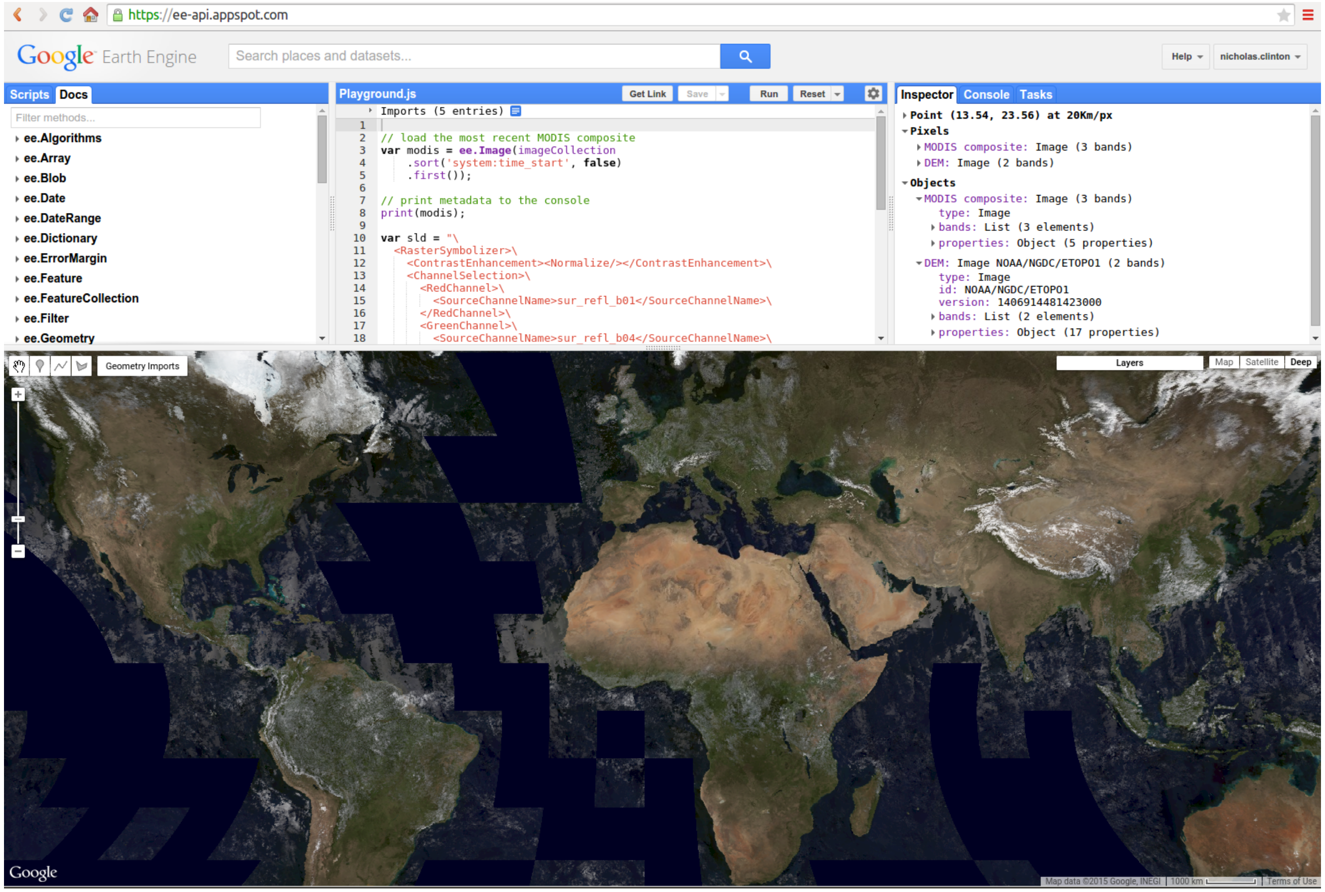
Understood by others

Expressed in a clear computational form

Interpreted by a “machine”

Answers and Insights

You will use this understanding in many situations that are not .py files and notebooks



```
Imports (5 entries)

// load the most recent MODIS composite
var modis = ee.Image(imageCollection
    .sort('system:time_start', false)
    .first());

// print metadata to the console
print(modis);

var sld = "\
  <RasterSymbolizer>\
    <ContrastEnhancement><Normalize/></ContrastEnhancement>\
    <ChannelSelection>\
      <RedChannel>\
        <SourceChannelName>sur_refl_b01</SourceChannelName>\
      </RedChannel>\
      <GreenChannel>\
        <SourceChannelName>sur_refl_b04</SourceChannelName>\
```

```
Point (13.54, 23.56) at 20Km/px
Pixels
  MODIS composite: Image (3 bands)
  DEM: Image (2 bands)
Objects
  MODIS composite: Image (3 bands)
    type: Image
    bands: List (3 elements)
    properties: Object (5 properties)
  DEM: Image NOAA/NGDC/ETOPO1 (2 bands)
    type: Image
    id: NOAA/NGDC/ETOPO1
    version: 1406914481423000
    bands: List (2 elements)
    properties: Object (17 properties)
```

SQL Review

SELECT

[ALL or DISTINCT] expressions over columns (map/reduce), optionally **AS** names

FROM

specification of table or join of tables

WHERE

conditional expression specifying rows in cols of tables

GROUP BY

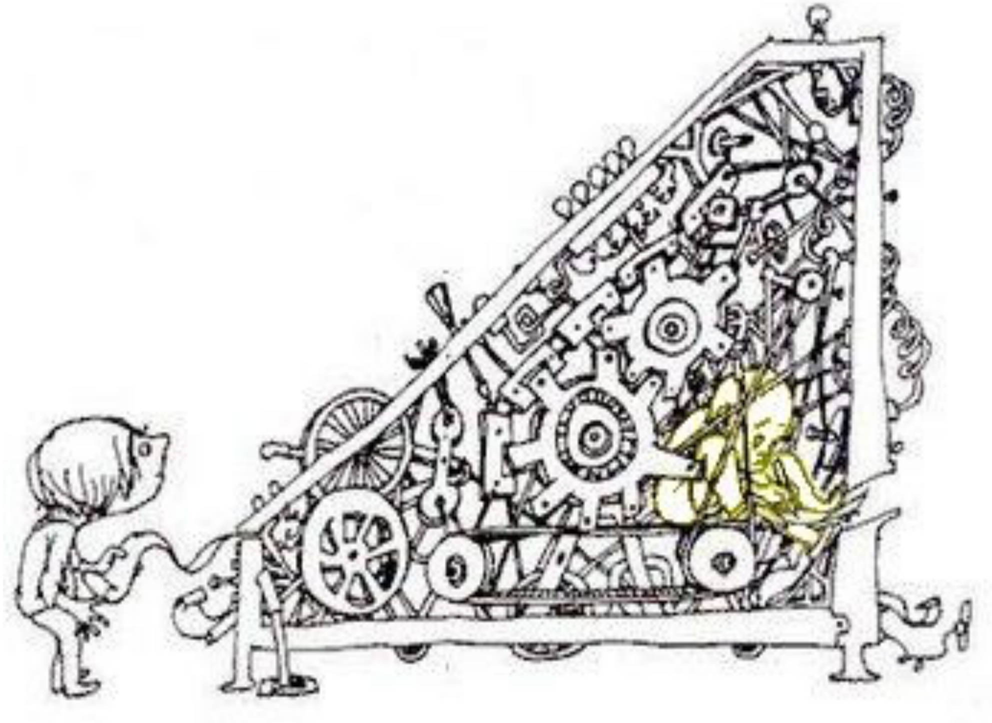
aggregation expression defining collections of rows in filtered cols of tables

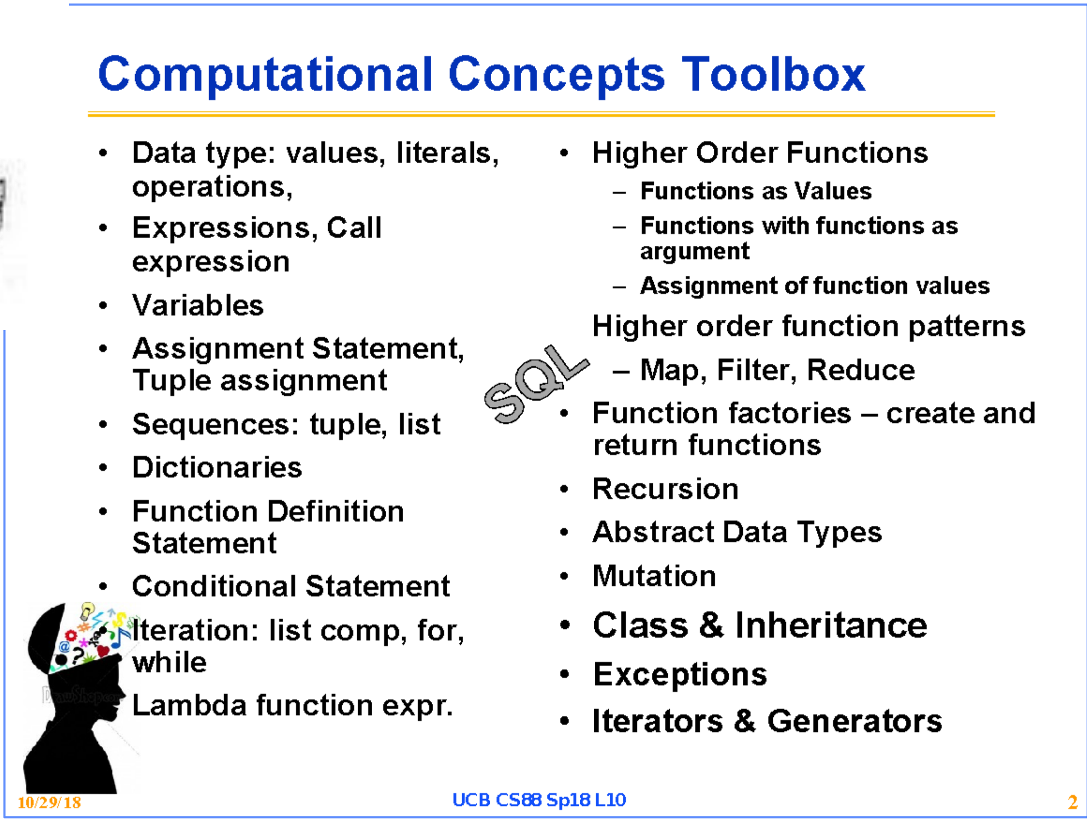
ORDER BY

expression on rows of filter cols defining order of result

;

How would you write a Python interpreter?





Computational Concepts Toolbox

- Data type: values, literals, operations,
- Expressions, Call expression
- Variables
- Assignment Statement, Tuple assignment
- Sequences: tuple, list
- Dictionaries
- Function Definition Statement
- Conditional Statement
- Iteration: list comp, for, while
- Lambda function expr.

SQL

- Higher Order Functions
  - Functions as Values
  - Functions with functions as argument
  - Assignment of function values

  Higher order function patterns
  - Map, Filter, Reduce
- Function factories – create and return functions
- Recursion
- Abstract Data Types
- Mutation
- **Class & Inheritance**
- **Exceptions**
- **Iterators & Generators**

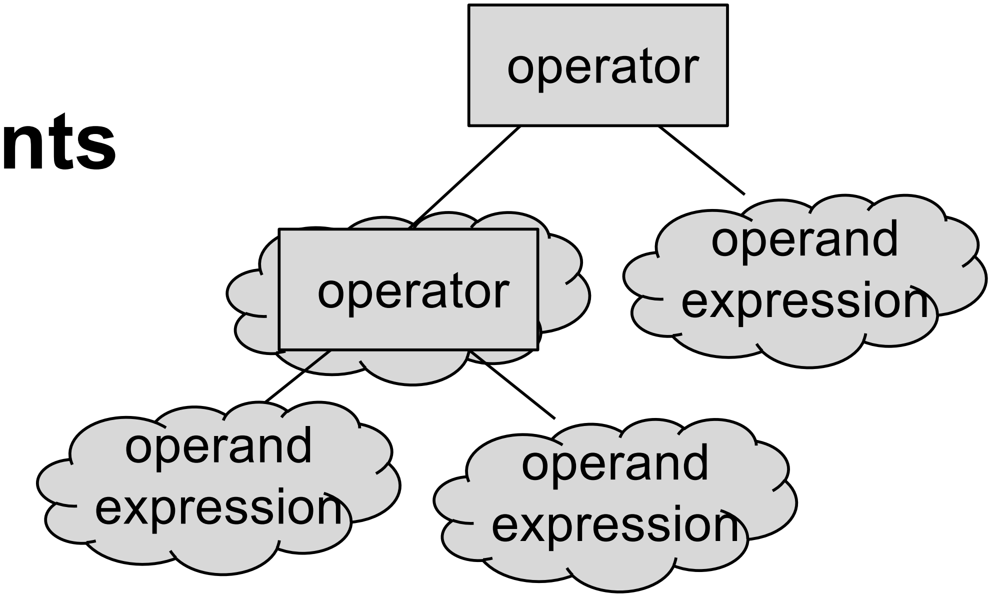
10/29/18 UCB CS88 Sp18 L10 2

What do you give to the interpreter?

- An Expression
- A sequence of Statements
- optionally followed by an expression

Basic Process

- Parse the input into logical pieces
- Expression
  - – Value or variable (leaves) – of a “type”
  - – Tree of operators and operand expressions
    - » .. * .. , .. + .. , …
    - » .. (..) , [..], lambda .. : .. , …
  - – Comprehensions
- Sequence of statements
  - – assignment
  - – def
  - – conditional
  - – iteration

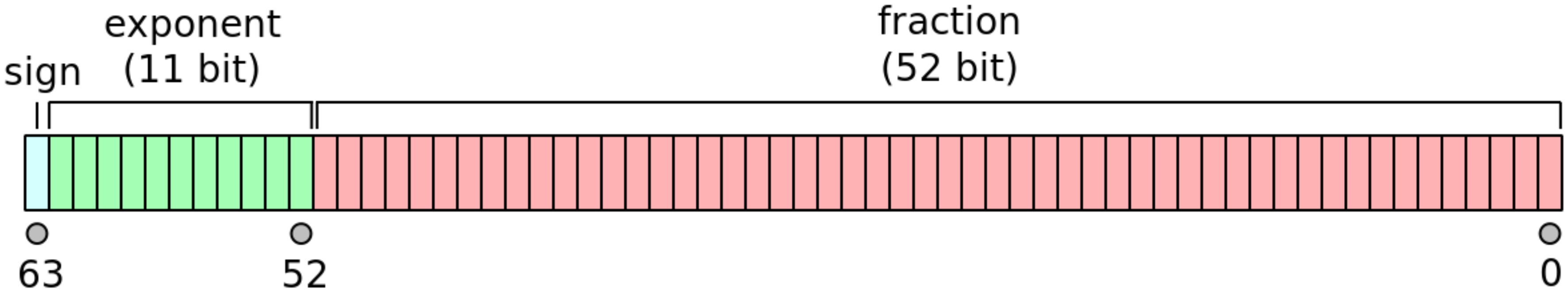


Values

- Primitive Value
  - int, float, boolean
- Complex Values
  - string, tuple, list, dict,
  - function, class
  - object, method
- Variable
  - Reference to a value

At the bottom it's a bunch of bits

- How many distinct things represented in *N* bits?
- 2^N - Think recursively
  - 2 "things" in 1 bit – {0,1}
  - Assume 2^{N-1} things in N-1 bits
  - 0 || {0, …, 2^{N-1} – 1} U 1 || {0, …, 2^{N-1} – 1}
- "word" is now (typically) 64 bits
  - Can represent 2^{64} (over 18 quintillion or 1.8×10^{19}) different values
- Addresses (unsigned ints): 0 .. $2^N – 1$
- Signed Integers: -2^{N-1} .. $2^{N-1} – 1$
- IEEE Float Point: $-1^S \times 1.f \times 2^{e-1023}$

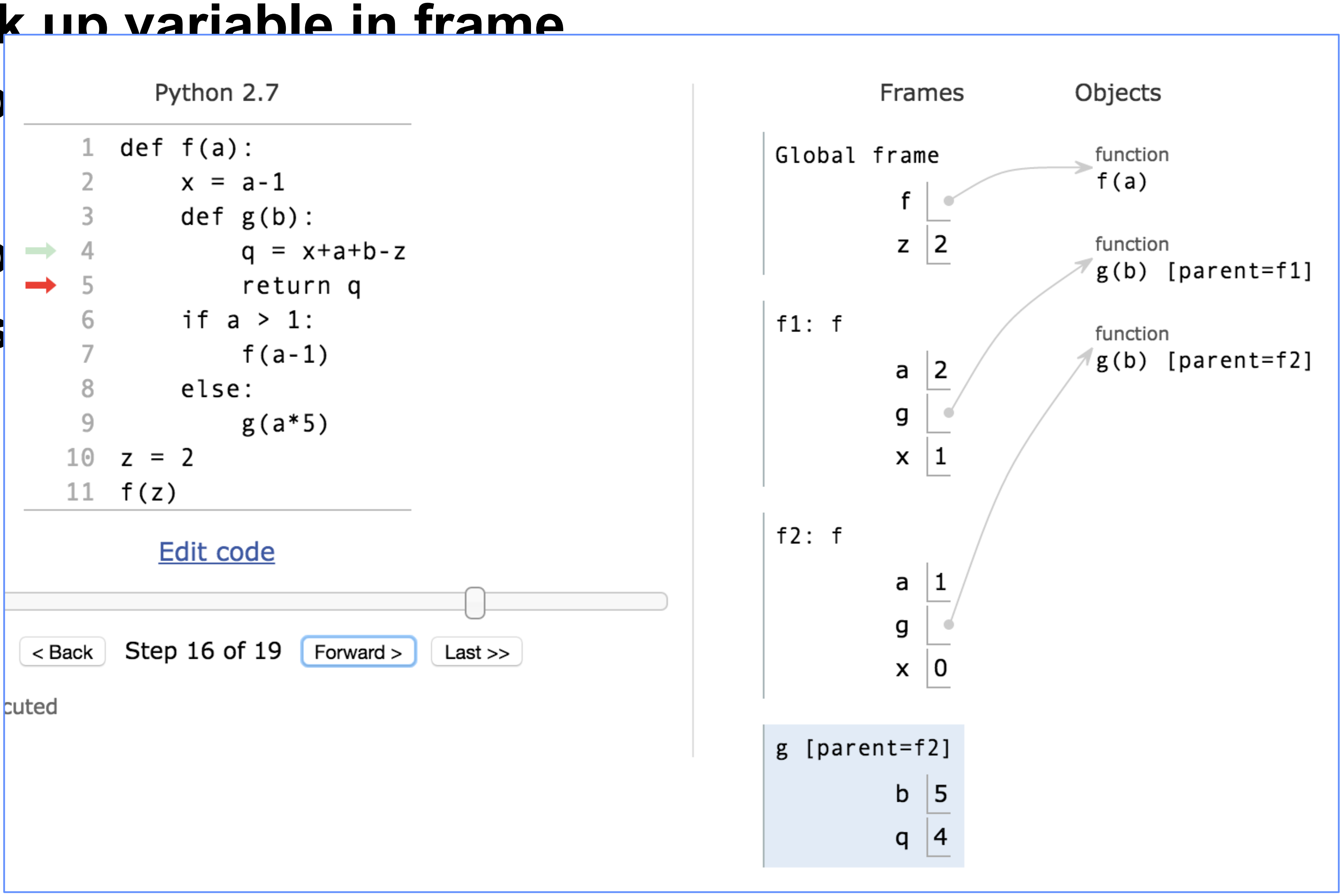


Variable

- Starting with current frame
- Look up variable in frame
- If not present, try parent frame, repeatedly
- Until global frame is reached
- If not found there
- Raise an exception

Variable

- Starting with current frame
- Look up variable in frame
- If no
- Unti
- If no
- Rais



Python 2.7

```
def f(a):
    x = a-1
    def g(b):
        q = x+a+b-z
        return q
    if a > 1:
        f(a-1)
    else:
        g(a*5)
z = 2
f(z)
```

Step 16 of 19

cuted

Global frame
f
z 2

f1: f
a 2
g
x 1

f2: f
a 1
g
x 0

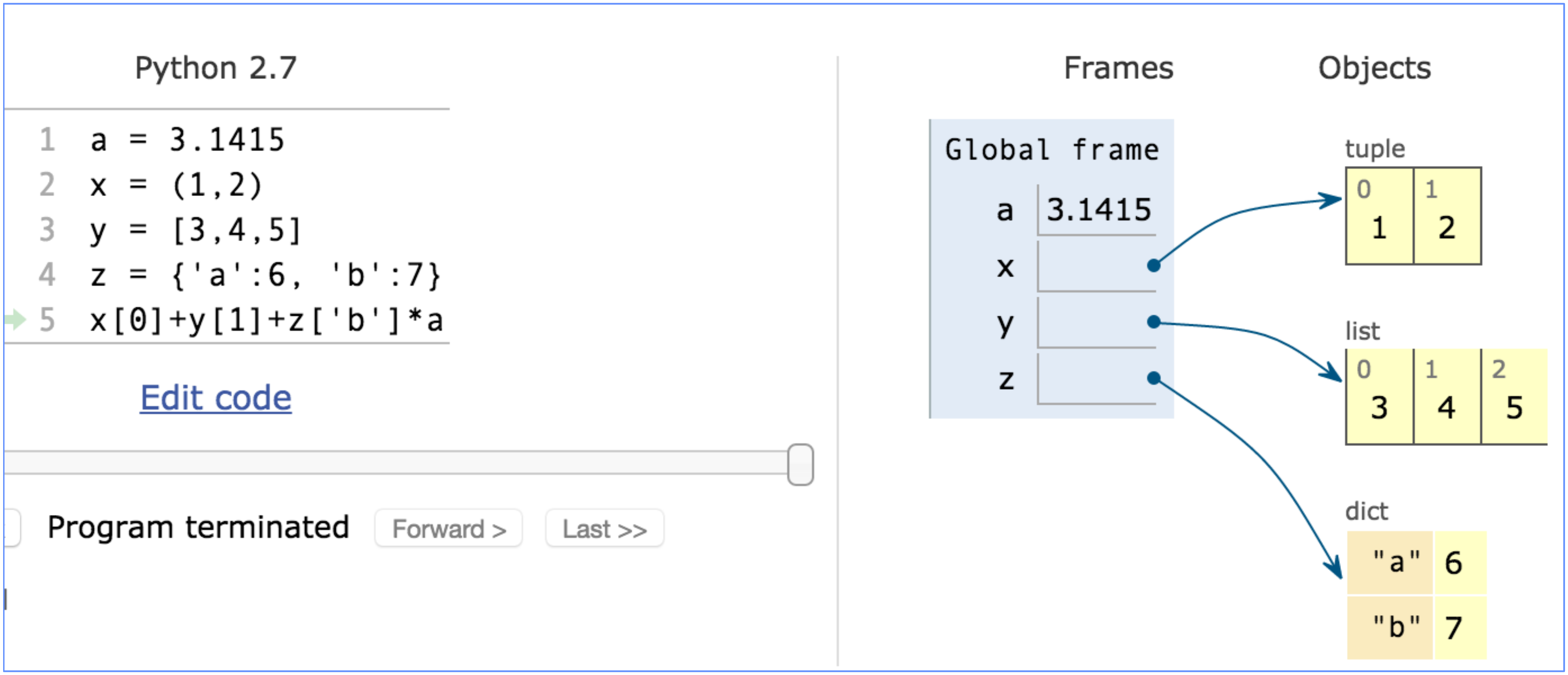
g [parent=f2]
b 5
q 4

function f(a)

function g(b) [parent=f1]

function g(b) [parent=f2]

Data Structures

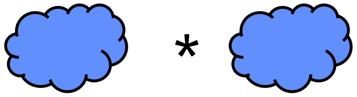


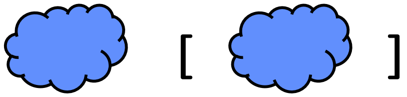
Python 2.7

```
a = 3.1415
x = (1,2)
y = [3,4,5]
z = {'a':6, 'b':7}
x[0]+y[1]+z['b']*a
```

Program terminated

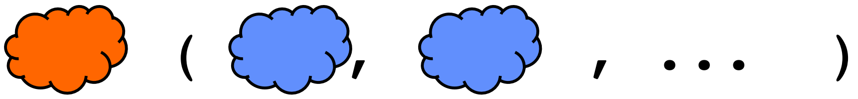
Operators





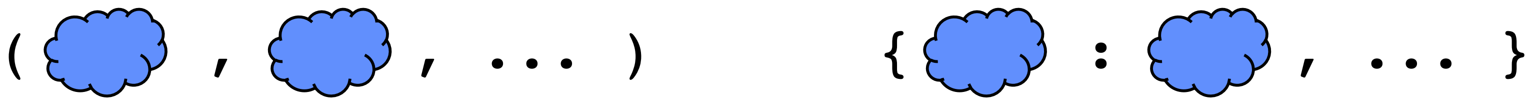
- Evaluate the operand expressions (recursively)
- Check the types of the resulting values to determine the operator for symbol
- If no valid combination, raise exceptions
- Apply operator to resulting values to produce result

Call Expressions



- Evaluate the operand expressions (recursively)
- Evaluate "function" expression to get function to apply
- This may involve function return values or "." or …
- Check that it is of function type
- If not, raise exception
- Apply function to resulting values to produce result

Built-in Data Structure Constructor



- Evaluate each of the index and value expressions
  - Or raise error
- Allocate storage to hold the data structure
- Fill in values at indices/Key
- Return a reference to the object

Comprehension Expression

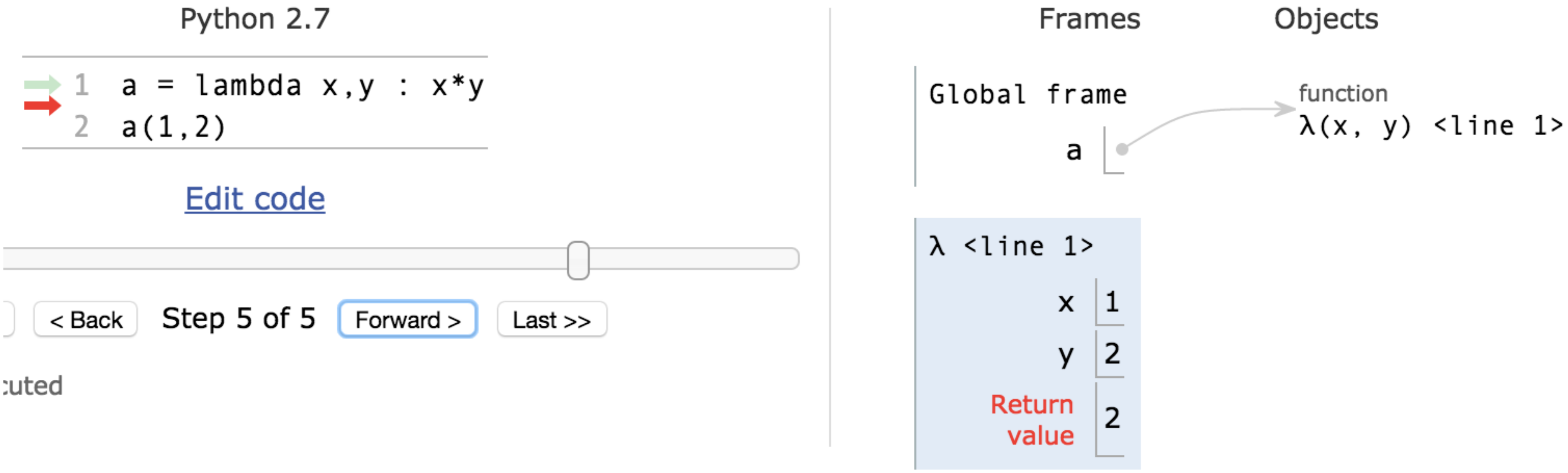
```
[ ☁   for <var tuple> in i-exp   ]
```

- Evaluate **iterable** expression
- For each element in iteration
- Bind var tuple to value tuple
- Evaluate ☁ with each of those variable bindings
- Construct resulting object and return reference to it

Lambda Expression

```
lambda <vars> : ☁
```

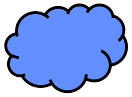
- **Construct a function object that evaluates expression ☁ in a frame with variables in `<vars>` bound to argument values and returning the result**
- **Return reference to the function object**



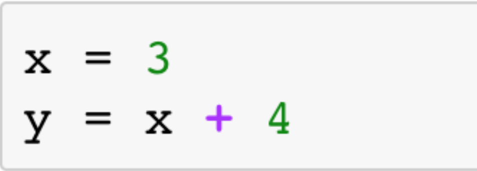
Python 2.7

```
a = lambda x,y : x*y
a(1,2)
```

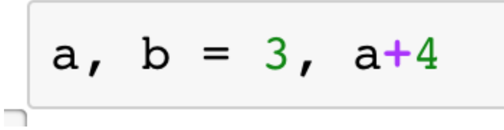
Assignment Statement

`<var list> =` 

- **Evaluate RHS expression to get value**
  - **Or raise and exception**
- **Locate LHS variable(s) in frame path**
- **For each variable**
- **If exists, set variable to expression value**
- **If not, create variable of name(s) `<var list>` in current frame**

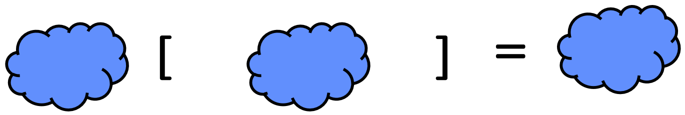


```
x = 3
y = x + 4
```



```
a, b = 3, a+4
```

Set operation



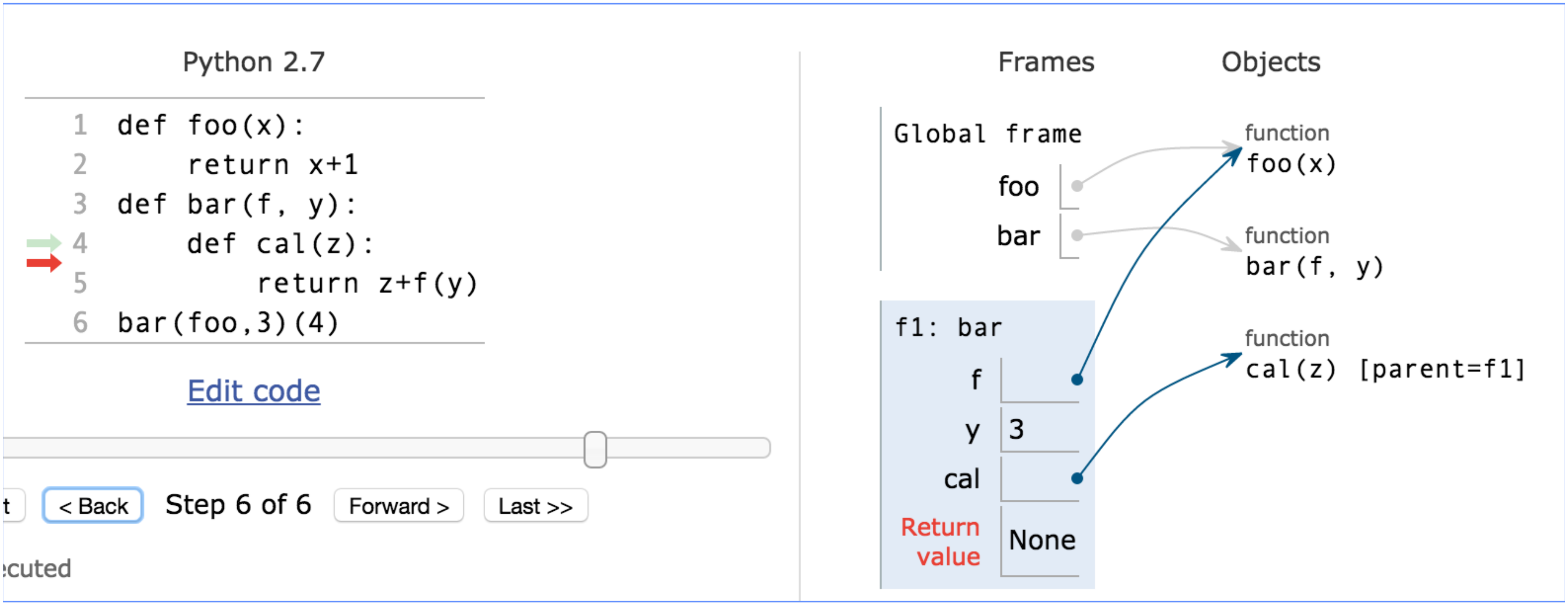
- Evaluate RHS to get value
  - Or raise and exception
- Evaluation LHS expressions to get object and index/key
  - Or raise exception
- Set obj [key] to expression value

Define Statement

```
def <fun name> ( <var list> ) :
    <suite of statements>
```

- Construct a function object to evaluate `<suite of statements>` in a frame with `<var list>` as local variable bound to argument expressions
- `return` statements evaluate expression in current frame and return it as result of the call expression
- Introduce `<fun name>` into current frame, assigned a reference to the function object

Define



Python 2.7

```
def foo(x):
    return x+1
def bar(f, y):
    def cal(z):
        return z+f(y)
bar(foo,3)(4)
```

Global frame
foo
bar

f1: bar
f
y 3
cal
Return value None

function foo(x)

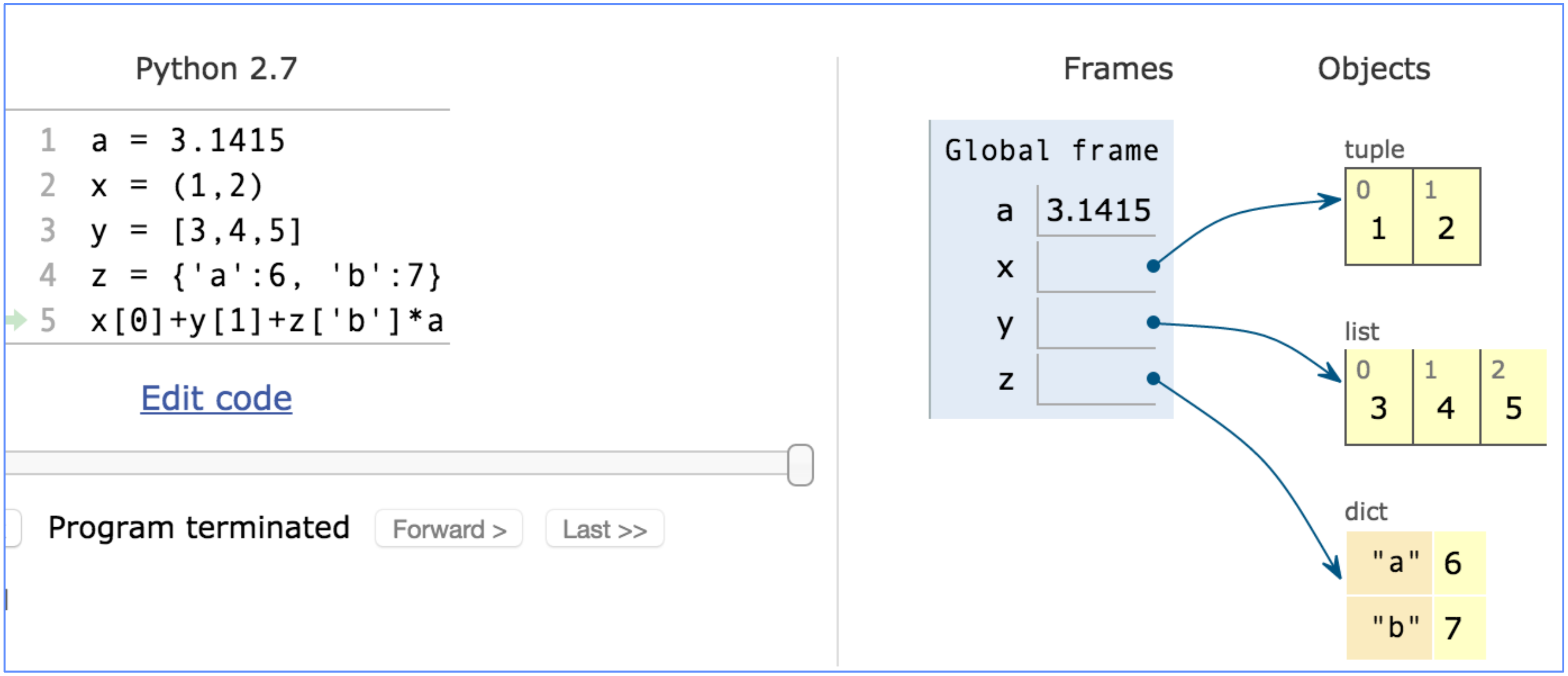
function bar(f, y)

function cal(z) [parent=f1]

Control Flow

Sequence of Statements

- Evaluate each statement in sequence
- Introducing new variables up updating objects with each



```
a = 3.1415
x = (1,2)
y = [3,4,5]
z = {'a':6, 'b':7}
x[0]+y[1]+z['b']*a
```

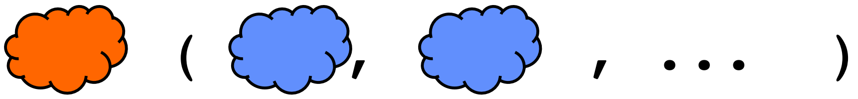
Conditional Statement

```
if ☁ :

    < true suite of statements >
else:
    <false suite>
```

- Evaluate ☁
- If it yields a truthy result, evaluate **`<true suite>`**
- Otherwise, if **`else:`** present, evaluate **`<false suite>`**

Call Expressions

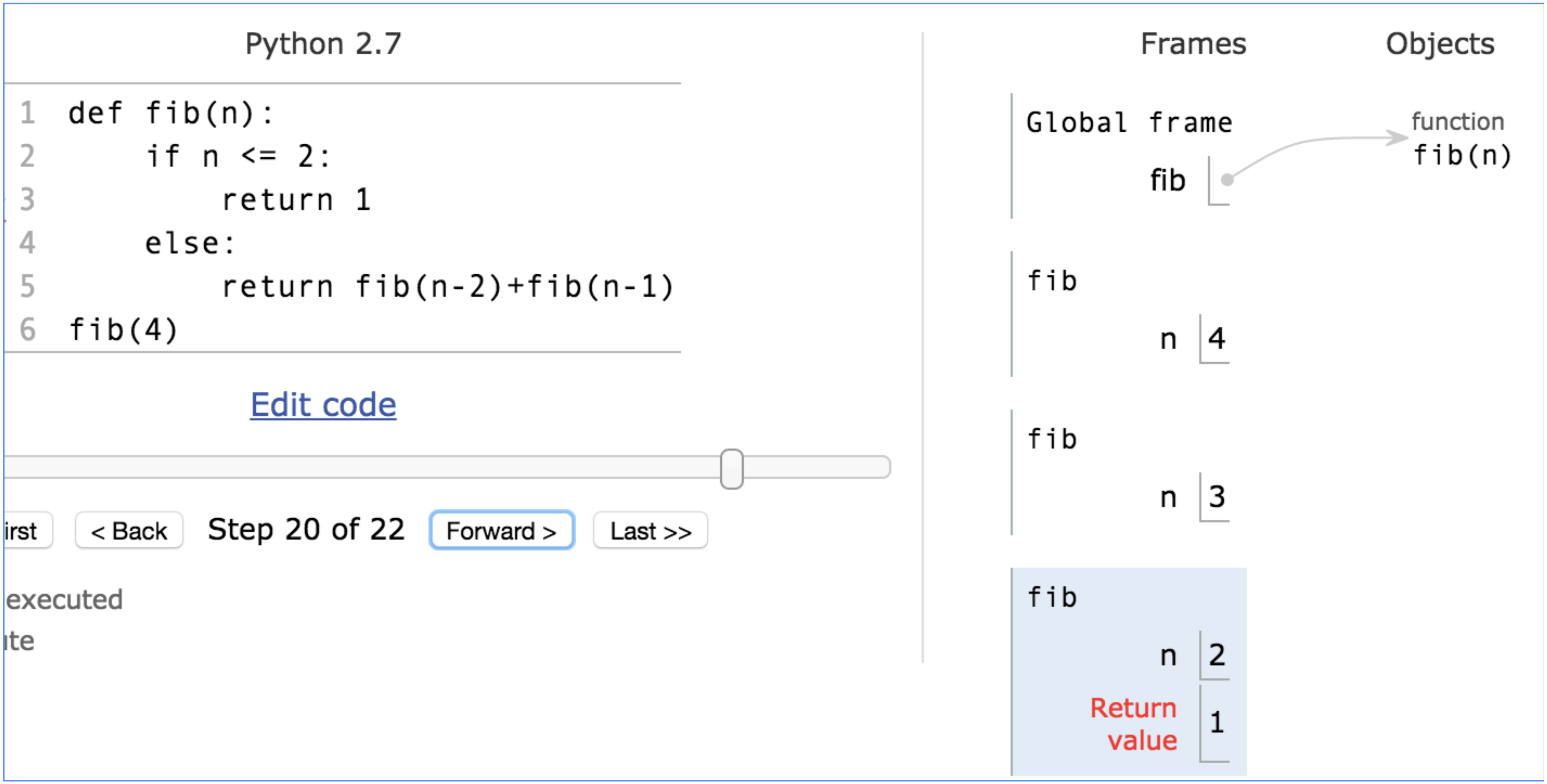


- Evaluate the operand expressions (recursively)
- Evaluate “function” expression to get function to apply
- This may involve function return values or “.” or …
- Check that it is of function type
- If not, raise exception
- *Apply function* to resulting values to produce result

Evaluate the statements within the function body

Functions plus conditionals …

- **Recursion**



Python 2.7

```
def fib(n):
    if n <= 2:
        return 1
    else:
        return fib(n-2)+fib(n-1)
fib(4)
```

```
Global frame
    fib  •──→  function
               fib(n)

fib
    n  4

fib
    n  3

fib
    n  2
    Return value  1
```

While Statement

```
while ☁ :

    < suite of statements >
else:
    < exit suite>
```

- **Repeatedly evaluate ☁**
- **If it yields a truthy result, evaluate `<suite>`**
- **Otherwise, if else: is present evaluate `<exit suite>`**
- **`continue` skips remain statements in suite**
- **`break` exits loop skipping `<exit suite>`**

For Statement

```
for <var list> in ☁   :

    < suite of statements >
else:
    < exit suite>
```

- Evaluate ☁ to get an iterable
- Repeatedly bind `<var list>` to next
- Evaluate `<suite>` with these bindings
- Until `StopIteration` is raised
- if else: is present evaluate `<exit suite>`

Try statement

```
try :
    < suite of statements >
except ☁ as  <var> :

    < except suite>
```

- **Evaluate suite of statements**
- **If exception is raised which matches ☁**
- **Evaluate except suite is var bound to exception object**

Class statement

```
class <classname> ( <inheritance> ):

    < suite of statements >
```

- **If present, evaluate the inheritance list to obtain a class object or class type.**
- **Create new namespace for classname**
- **Evaluate <suite> in a new execution frame using a newly created namespace and global namespace**
  - **Typically sequence of define statements**
- **`self` in define for methods, `self` otherwise for object attributes**
- **vars in class namespace for class attributes**
- **Return resulting class object**

. operator

```
☁ . <var>
```

- **References <var> in namespace of ☁**

with statement

```
with ☁ as <var> [, more ] :
    < suite of statements >
```

- **Evaluate suite of statements with vars bound to results of corresponding ☁**

Comprehension expressions

```
[ 🟦 for <var list> in 🟦          ]

[ 🟦 for <var list> in 🟩 if 🟥 ]
```

- **Iteratively,**
  - **Evaluate next 🟩**
  - **If present, evaluate 🟥 on it**
  - **Evaluate 🟦**
- **until stop_iter exception**
  - **Collect all resulting values into result object**

Software Design Patterns

- Higher Order Functions
- Recursion
- Data Parallel – Map-Reduce
- Abstract Data Types
  - Constructors, Selectors, Actions
- Object Oriented Programming
  - Encapsulation of behavior
- Iterators and Generators
  - Classes with __iter__ and __next__
  - `yield` statement

Uses of Computational Thinking

- Computational concepts model the world. Programming languages are mathematical formalisms just like any other: linear algebra, differential equations, statistics…

- Plus: Automatic verification of the model.

More CS:

- CS61b: More programming
- CS61c: Machine architecture (how the bits are moved)

So now …

Go model and change the world …