Blocks and Contexts: Exploring Scientific Algorithms

Eric Jones

SciPy 07

August 17, 2007

Scientists often know how to model their problems in software.

Its exploring them that is hard.

How do we make it easier?

Fortran Example

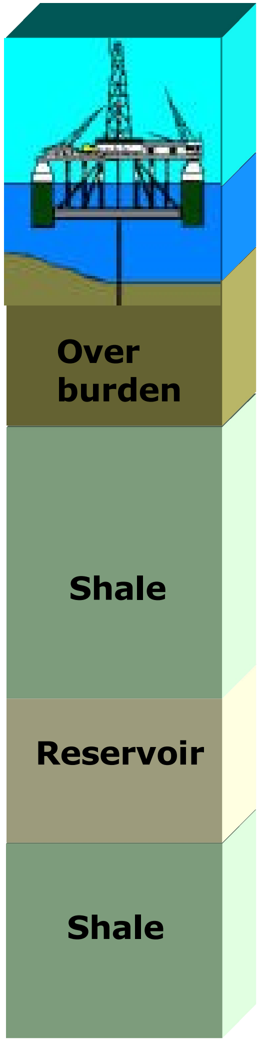
```
 PROGRAM ONE_D_MOTION
C Program for the motion of a particle subject to an external
C force f(x) = -x. The position and velocity of the particle
C are written out at every 500 steps.
      PARAMETER (N=10001,IN=500)
      REAL T(N),V(N),X(N)
C Assign constants, initial position, and initial velocity
      PI   = 4.0*ATAN(1.0)
      DT   = 2.0*PI/FLOAT(N-1)
      X(1) = 0.0
      T(1) = 0.0
      V(1) = 1.0
C Recursion for position and velocity at later time
      DO      100  I = 1, N-1
        T(I+1) = DT*I
        X(I+1) = X(I)+V(I)*DT
        V(I+1) = V(I)-X(I)*DT
  100 CONTINUE
C Write the position and velocity every 500 steps
      WRITE (6,999) (T(I),X(I),V(I),I=1,N,IN)
      STOP
```

From: "An Introduction to Computational Physics,“ by Tao Pang

Typical Scientific Code

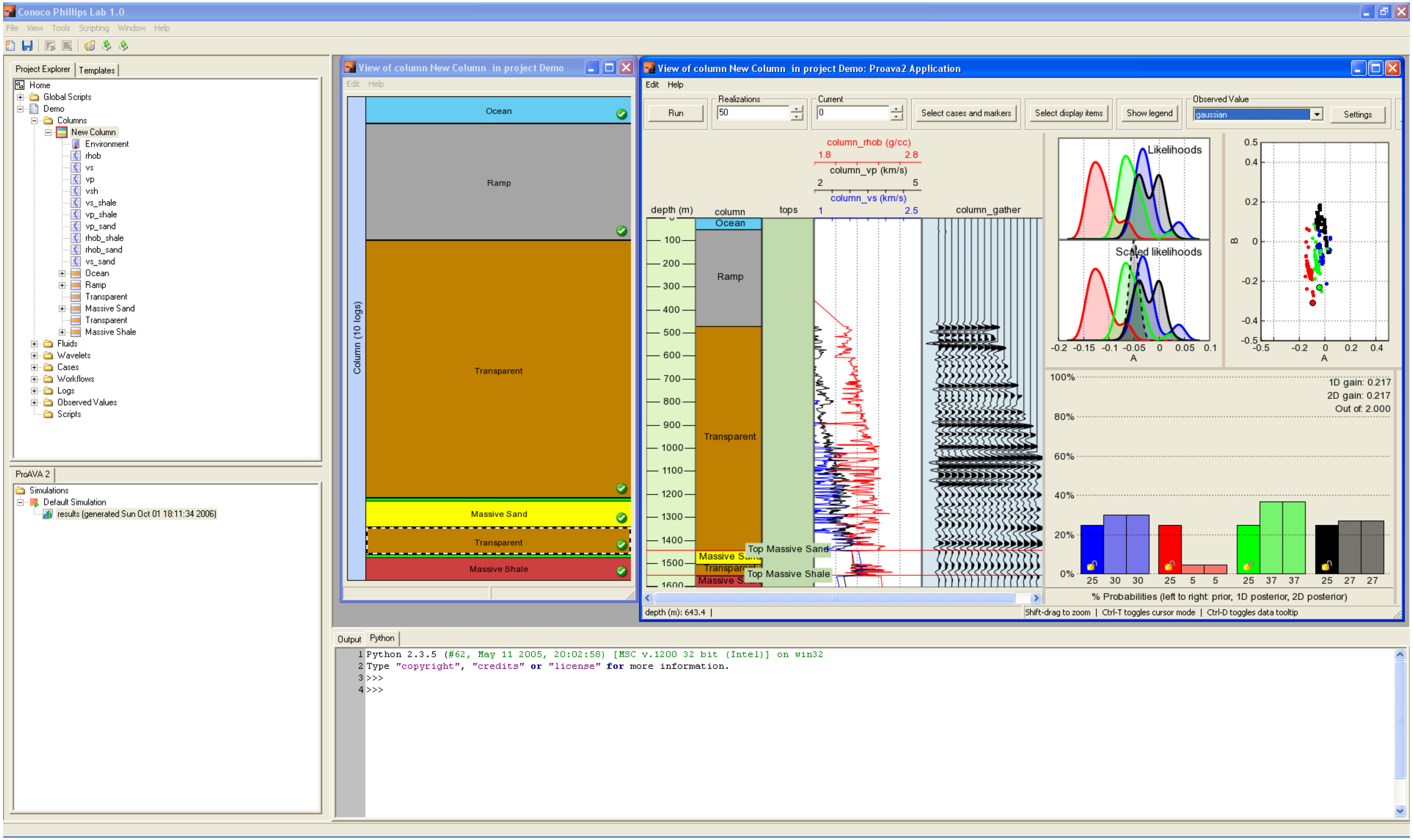
```
from numpy import arange, ravel, minimum
from scipy.integrate import odeint

# Define functions.
def growth_structure(y,t, k1, k2, gammadot):
    res = -k1*gammadot*y+k2*(1.0-y)
    return res

def growth(t, y, k1, k2, gammadot, gammaC0, G, k, m, n,):
    visc=(1.0-y) * k * gammadot**n
    gammaE = gammadot*t
    gammaC = gammaC0 * y**m
    elastic = y * G * minimum(gammaE, gammaC)
    total = visc + elastic
    return visc, elastic, total

# Set up algorithm parameters and intial guesses...
k=400.0; n=0.7; gammadotgrowth=20.0; gammadotrelax=0.0
gammaC0 =0.5; m=-0.33; G=25000.0; k1=2.0; k2=1.0
y0 = 1.0
x1 = arange(0.0, .5, 0.005)

# The actual calculation.
y1 = odeint(growth_structure, y0, x1, args=(k1,k2,gammadotgrowth))
y1 = ravel(y1)
viscous, elastic, total = growth(x1, y1, k1, k2, gammadotgrowth,
                                 gammaC0, G, k, m, n)
```

Complex Problems, Simple Algorithms



```
# ocean
vp = 1.5
vs = 1e-5
rhob = 1.05

# interpolation region


# bulk shale
xsh = xsh_shale1
vp, vs, rhob = backus_avg(xsh, vp_sh, vs_sh, rhob_sh,
                          vp_s, vs_s, rhob_s)

# fining upward
xsh = linear([top, base], [max_xsh, min_xsh], len(vp_sh))
vp, vs, rhob = backus_avg(xsh, vp_sh, vs_sh, rhob_sh,
                          vp_s, vs_s, rhob_s)

# bulk shale
xsh = xsh_shale2
vp, vs, rhob = backus_avg(xsh, vp_sh, vs_sh, rhob_sh,
                          vp_s, vs_s, rhob_s)
```

Stochastic Modeling Tool



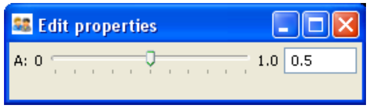
```
Python 2.3.5 (#62, May 11 2005, 20:02:58) [MSC v.1200 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
>>>
>>>
```

Analysis of Models

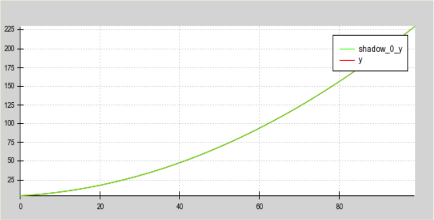
Scientific Model

```
a=1
b=2
c=3
y = a*x**2+b*x+c
```

"What-if" analysis:

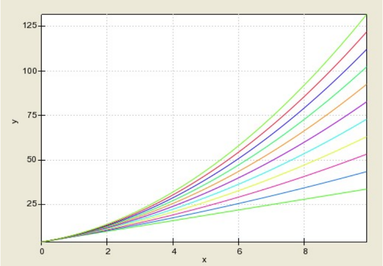


A: 0 1.0 0.5



Parametric Studies:

low 0.0 high 1.0 step 0.1



Monte Carlo:

a mean 0.0 std 1.0

Inversion: Given y, invert for a, b, and c.

Blocks and Contexts

Code Block

```
a=1
b=2
c=3
y = a*x**2+b*x+c
```

Code blocks are a set of executable instructions.

Context

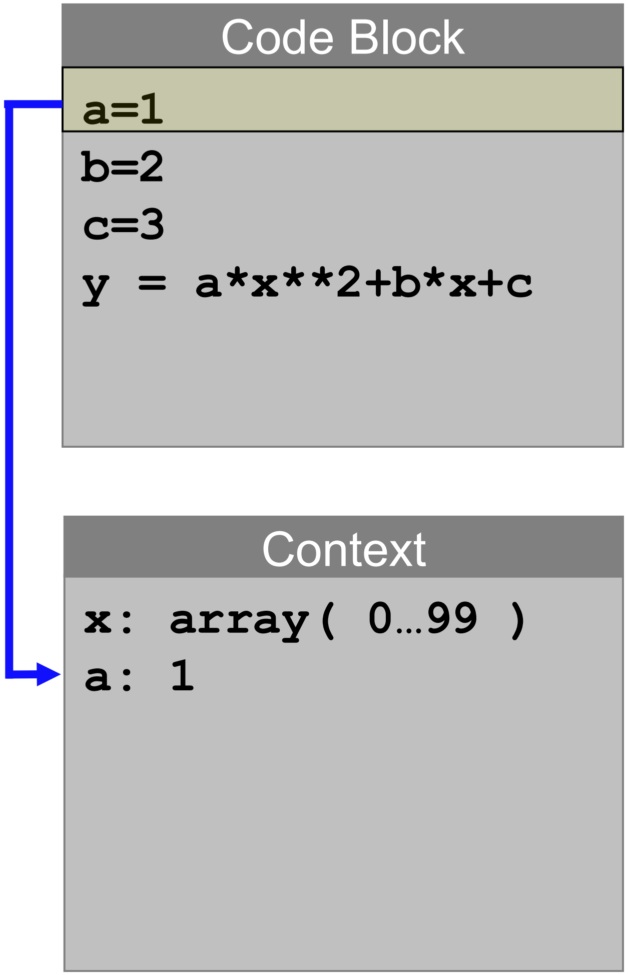
```
x: array( 0…99 )
```

Contexts or “namespaces” is a mapping of names to values.

Python Code Execution

```
# What really happens when you execute the following code?
a = 1
b = 2
c = 3
y = a*x**2+b*x+c
```

Blocks and Contexts



Code Block

```
a=1
b=2
c=3
y = a*x**2+b*x+c
```

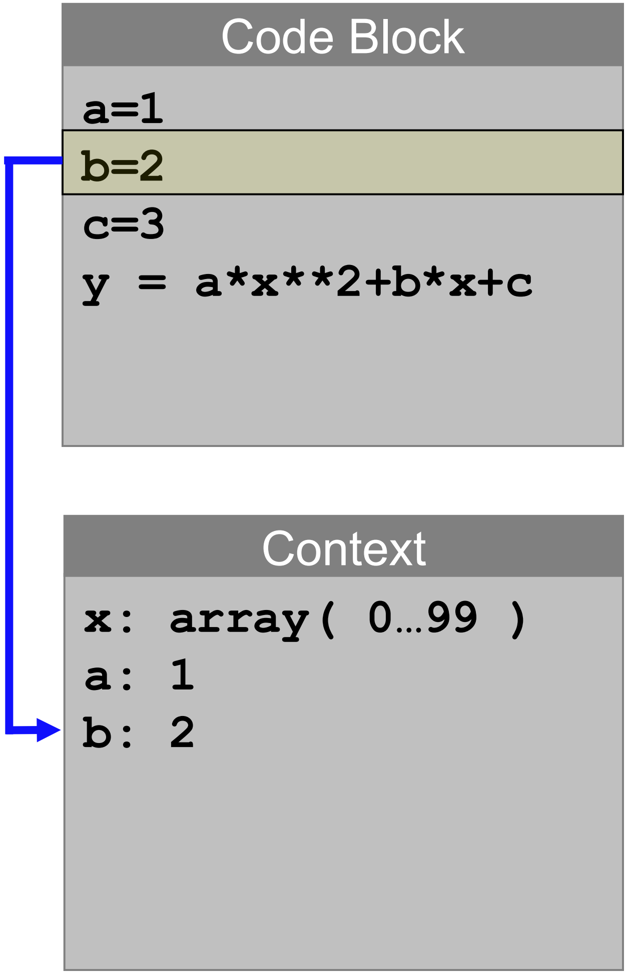
Context

```
x: array( 0…99 )
a: 1
```

Code blocks are a set of executable instructions.

Contexts or “namespaces” is a mapping of names to values. Here, we’ll assume we started with ‘x’ already in the namespace.

Blocks and Contexts



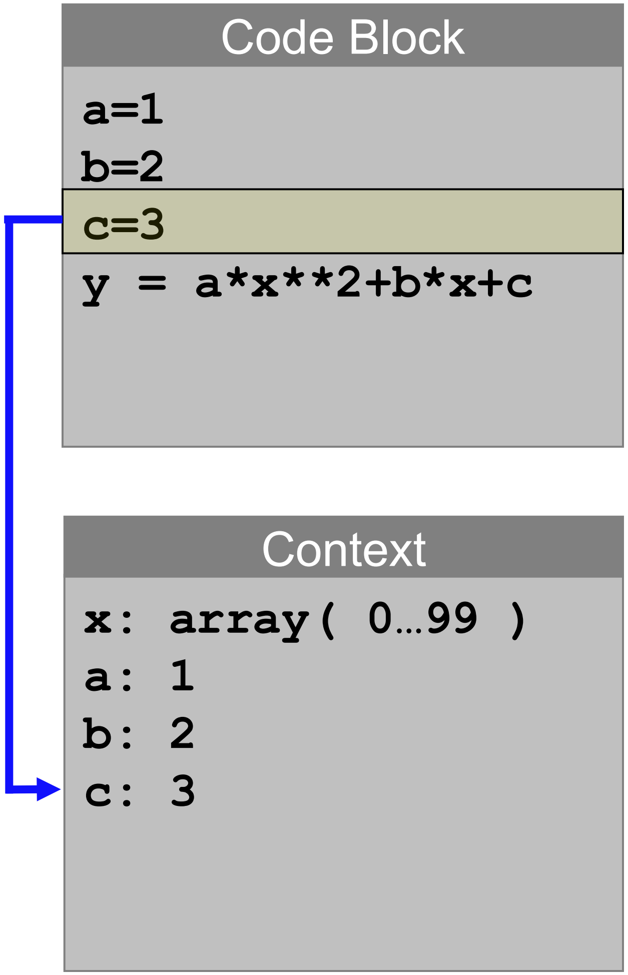
```
a=1
b=2
c=3
y = a*x**2+b*x+c
```

```
x: array( 0…99 )
a: 1
b: 2
```

Code blocks are a set of executable instructions.

Contexts or “namespaces” is a mapping of names to values.

Blocks and Contexts



Code Block

```
a=1
b=2
c=3
y = a*x**2+b*x+c
```

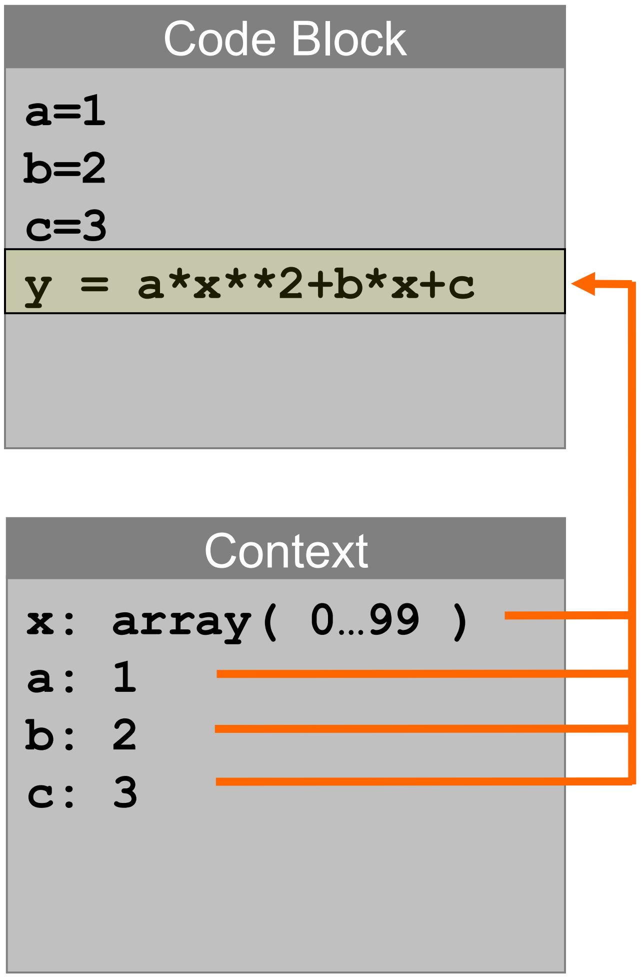
Context

```
x: array( 0…99 )
a: 1
b: 2
c: 3
```

Code blocks are a set of executable instructions.

Contexts or “namespaces” is a mapping of names to values.

Blocks and Contexts



Code Block

```
a=1
b=2
c=3
y = a*x**2+b*x+c
```

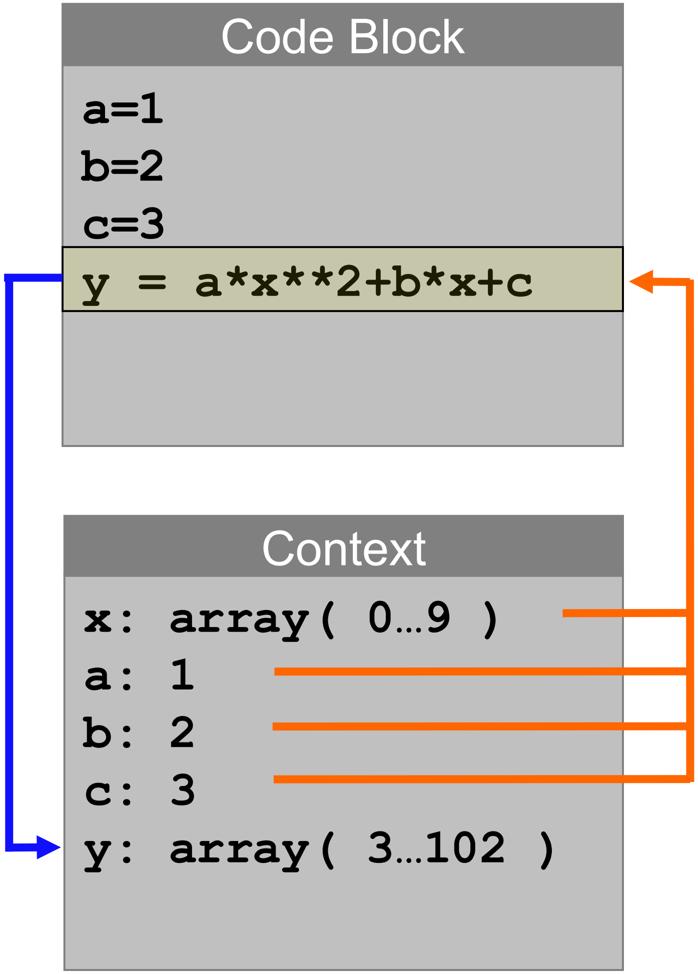
Context

```
x: array( 0…99 )
a: 1
b: 2
c: 3
```

Code blocks are a set of executable instructions.

Contexts or “namespaces” is a mapping of names to values.

Blocks and Contexts



Code blocks are a set of executable instructions.

Contexts or “namespaces” is a mapping of names to values.

exec in a dictionary

Code Block

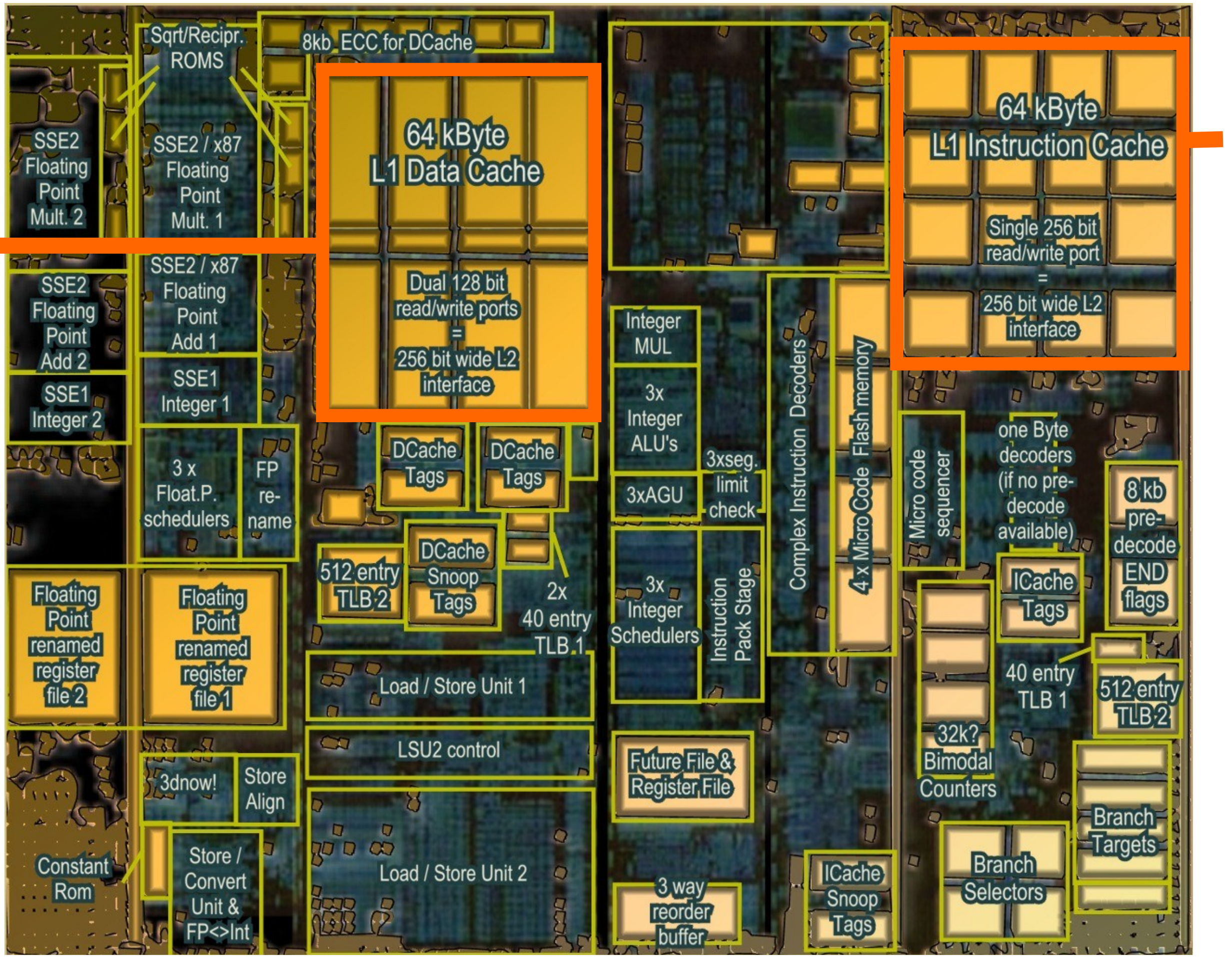
```
a=1
b=2
c=3
y = a*x**2+b*x+c
```

Context

```
x: array( 0…9 )
a: 1
b: 2
c: 3
y: array( 3…102 )
```

```
# set up context and load with ‘x’
>>> context = {}
>>> context['x'] = arange(10.)

# execute code block in context
>>> code = '''
a = 1
b = 2
c = 3
y = a*x**2+b*x+c
'''
>>> exec code in {}, context

# y was computed and put in context.
>>> context['y']
array(3...102)
```

Single Core for AMD's upcoming Quad-core processor


Sqrt/Recipr. ROMS
8kb ECC for DCache
64 kByte L1 Data Cache
Dual 128 bit read/write ports = 256 bit wide L2 interface
64 kByte L1 Instruction Cache
Single 256 bit read/write port = 256 bit wide L2 interface
SSE2 Floating Point Mult. 2
SSE2 / x87 Floating Point Mult. 1
SSE2 Floating Point Add 2
SSE2 / x87 Floating Point Add 1
SSE1 Integer 2
SSE1 Integer 1
3 x Float.P. schedulers
FP re-name
DCache Tags
DCache Tags
DCache Snoop Tags
512 entry TLB 2
2x 40 entry TLB 1
Integer MUL
3x Integer ALU's
3xAGU
3xseg. limit check
3x Integer Schedulers
Instruction Pack Stage
Complex Instruction Decoders
4 x Micro Code Flash memory
Micro code sequencer
one Byte decoders (if no pre-decode available)
8 kb pre-decode END flags
ICache Tags
40 entry TLB 1
512 entry TLB 2
32k? Bimodal Counters
Branch Targets
Branch Selectors
ICache Snoop Tags
Floating Point renamed register file 2
Floating Point renamed register file 1
Load / Store Unit 1
LSU2 control
3dnow!
Store Align
Future File & Register File
Constant Rom
Store / Convert Unit & FP<>Int
Load / Store Unit 2
3 way reorder buffer

Data Cache

Instruction Cache

Contexts with Events

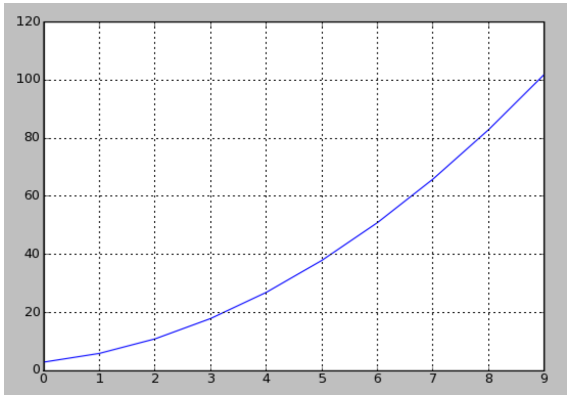
Code Block

```
a=1
b=2
c=3
y = a*x**2+b*x+c
```

Context

```
x: array( 0…9 )
a: 1
b: 2
c: 3
y: array( 3…102 )
```

Events Fire when data changes



Updating Data View

Contexts with Events

```
# set up context and load with ‘x’
>>> context = DataContext()
>>> context['x'] = arange(10.)

# hook up listener for changes to context.
>>> def printer(event):
...       print 'added:', event.added
>>> context.on_trait_change(printer, 'items_modified')

# execute code block in context
>>> code = '''
a = 1
b = 2
c = 3
y = a*x**2+b*x+c
'''
>>> exec code in {}, context
added: ['a']
added: ['b']
added: ['c']
added: ['y']
```

Text printed by the listener defined above.

Interacting with a Variable

Suppose we want to do “what-if” analysis to see how changes to ‘a’ affect our model.

Original Block

```
a=1
b=2
c=3
y = a*x**2+b*x+c
```

Dependency Analysis:
Extract sub-block
that is affected by **`a`**

What-if Block

```
y = a*x**2+b*x+c
```

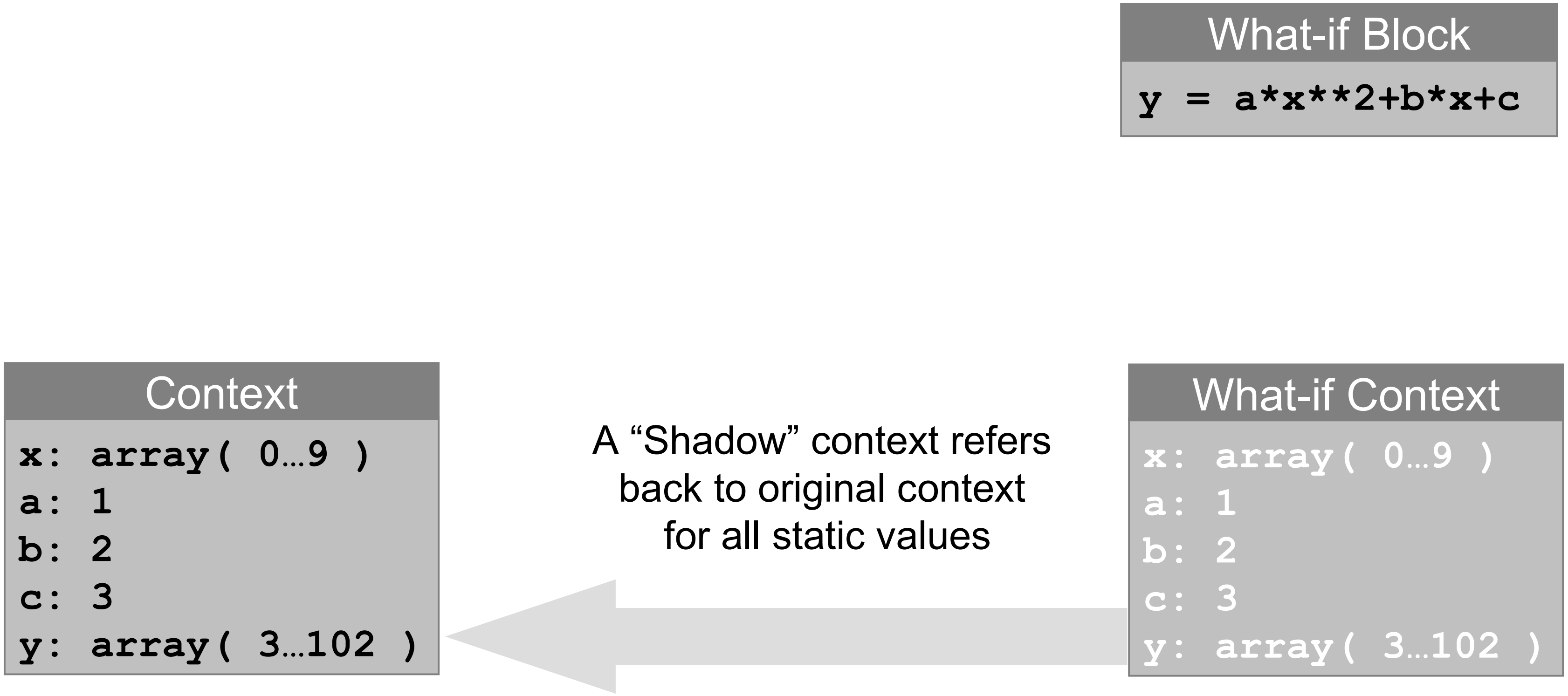
Context

```
x: array( 0…9 )
a: 1
b: 2
c: 3
y: array( 3…102 )
```

Dependency Analysis

```
# Create a “Block” that represents/analyzes code
>>> code = '''
a = 1
b = 2
c = 3
y = a*x**2+b*x+c
'''
>>> block = Block(code)

# Calculate the sub-block affected by updates to x
>>> sub_block = block.restrict(inputs=['a'])
>>> print compiler_unparse.unparse(sub_block.ast)
y = a*x**2+b*x+c

>>> sub_block.inputs
set(['a', 'x', 'c', 'b'])

>>> sub_block.outputs
set(['y'])
```

Interacting with a Variable



What-if Block

```
y = a*x**2+b*x+c
```

Context

```
x: array( 0…9 )
a: 1
b: 2
c: 3
y: array( 3…102 )
```

A “Shadow” context refers back to original context for all static values

What-if Context

```
x: array( 0…9 )
a: 1
b: 2
c: 3
y: array( 3…102 )
```

Interacting with a Variable

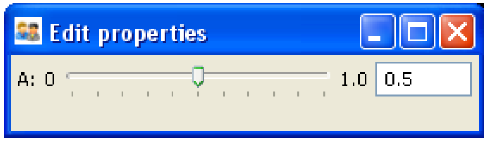
What-if Block

```
y = a*x**2+b*x+c
```

Changes to Block inputs
re-computes outputs.

Context

```
x: array( 0…9 )
a: 1
b: 2
c: 3
y: array( 3…102 )
```

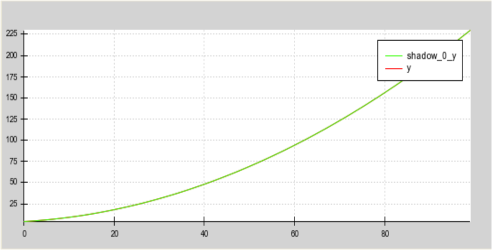


A: 0 1.0 0.5

“What-if” analysis
on variable ‘a’

What-if Context

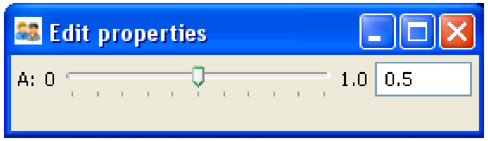
```
x: array( 0…9 )
a: 0.5
b: 2
c: 3
y: array(3…61.5)
```



Plots update to show
new values over old

Implementing Shadow Contexts

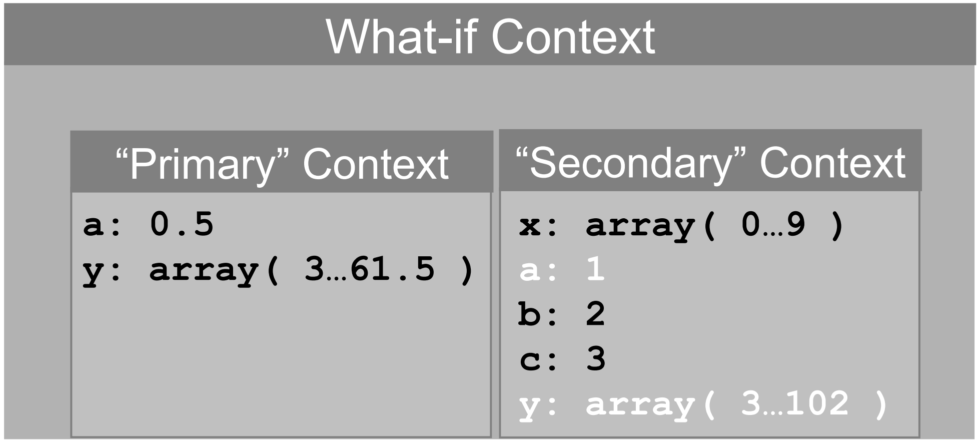
- Writes always happen to the Primary Context.
- Reads first try in Primary Context. If that fails, they try the Secondary Context.



A: 0 1.0 0.5

What-if Block

```
y = a*x**2+b*x+c
```



What-if Context

“Primary” Context

```
a: 0.5
y: array( 3…61.5 )
```

“Secondary” Context

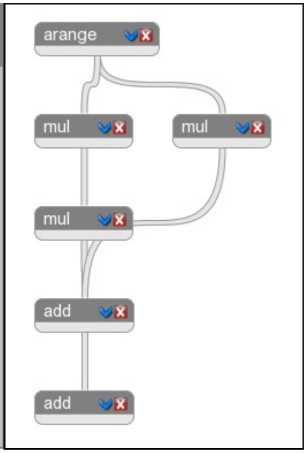
```
x: array( 0…9 )
a: 1
b: 2
c: 3
y: array( 3…102 )
```

Functions vs. Data

- Function Context accepts only Functions.
- Data Context holds everything else.
- This prevents the data context from getting cluttered.

Block

```
from my_operator import add, mul
from numpy import arange
x = arange(0,10,.1)
x1 = mul(x,x)
t1 = mul(a,x1)
t2 = mul(b, x)
t3 = add(t1,t2)
y = add(t3,c)
```



Canvas Context

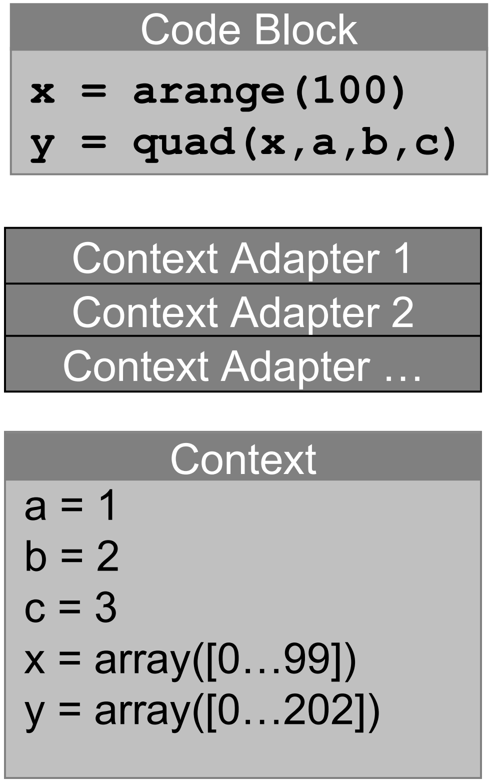
Function Context

```
arange
add
mul
```

Data Context

```
a: 0.5
b: 3.0
c: 4.0
x: array( 0…9 )
...
```

Context Adapters



Code Block

```
x = arange(100)
y = quad(x,a,b,c)
```

Context Adapter 1

Context Adapter 2

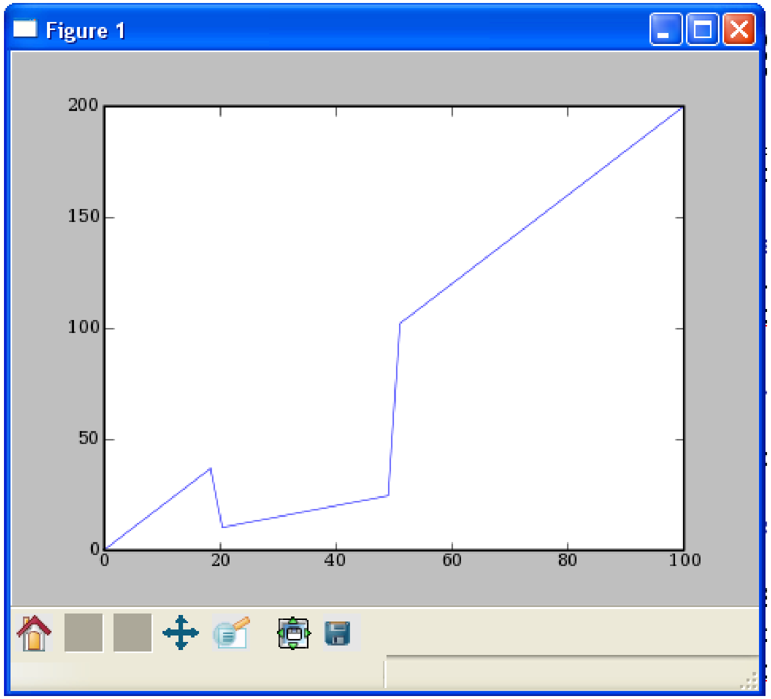
Context Adapter …

Context

a = 1
b = 2
c = 3
x = array([0…99])
y = array([0…202])

A Masking Adapter

```
context = AdaptedDataContext(context=DataContext())

# Add some depth values as data in the context.
depth=linspace(0,100)
context.update(depth=depth)

# Calculate a pressure based on depth.
code = "pressure = depth*2.0"
exec code in globals(), context

# "Mask" context so that it only a
ffects certain ranges of data.
mask=(20.0<=depth) & (depth<=50.0)
adapter = MaskingAdapter(mask=mask)
context.push_adapter(adapter)

# Calculate new pressures for masked values.
code = "pressure = depth/2.0"
exec code in globals(), context

# Unmask the context
context.pop_adapter()
```



Using with inside a context.

```
# Calculate pressure at depth using
# a simple formula.
depth=linspace(0,100)
pressure = depth*2.0

# "Mask" context so that code only
# affects certain ranges of data.
with Mask((20.0<=depth) & (depth<=50.0)):
    # In this region, use a different
    # formula for pressure.
    pressure = depth/2.0
```

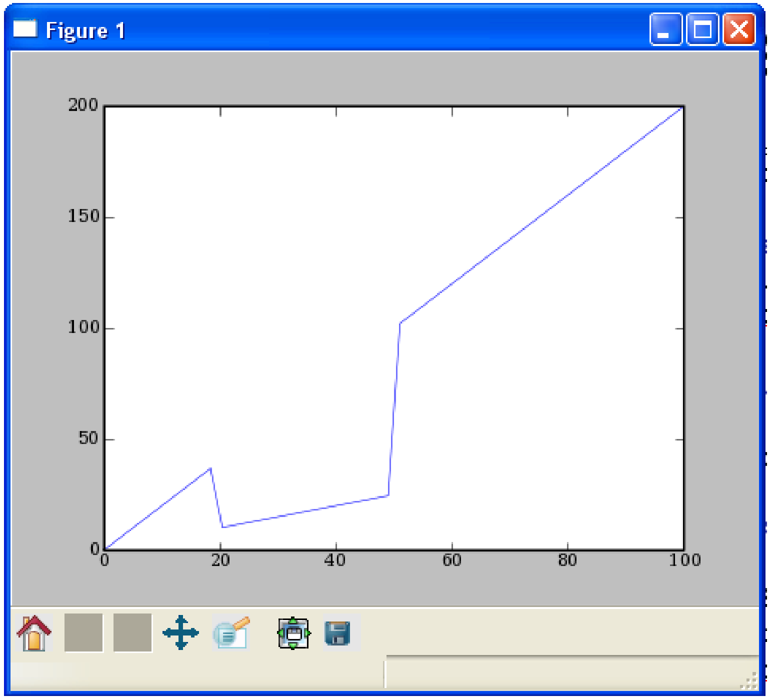


Figure 1
200
150
100
50
0
0
20
40
60
80
100