Python for Scientific Applications

An overview of modern software design for scientific applications

Michael Aivazis

Ay/Bi 199
January 2009

Programming paradigms

- A very active area of research
  - *dozens of languages and runtime environments over the last 40 years*
- The survivors:
  - *Procedural programming*
    - *Structured programming*
  - *Functional programming*
  - *Object oriented programming*
- Current areas of reasearch
  - *Component oriented programming*
  - *Aspect programming*
- Language constructs
  - *reflect an approach to computing*
  - *shape what is easily expressible*

Sources of complexity

- Project size:
  - *asset complexity: number of lines of code, files, entry points*
  - *dependencies: number of modules, third-party libraries*
  - *runtime complexity: number of objects types and instances*
- Problem size:
  - *number of processors needed, amount of memory, cpu time*
- Project longevity:
  - *life cycle, duty cycle*
  - *cost/benefit of reuse*
  - *managing change: people, hardware, technologies*
- User interfaces
- Locality of needed resources

- Turning *craft* into: *science, engineering, … art*

Managing complexity

- Understand the “client”
  - *be explicit about requirements, limitations (scope)*
  - *broaden your notion of user interface*
- Code architecture
  - *given a problem*
    - *all languages are equal*
    - *some are more equal than others*
  - *flexible programming and runtime environments*
  - *design for change; but keep it real*
- Sensible software engineering practices
  - *document the design, not just the implementation*
  - *source control, issue tracking, documentation*
  - *build, release and deployment strategies*

A quick introduction to Python

- Resources
- Interacting with the Python interpreter
  - *Interactive sessions*
- Overview of the Python language
  - *The focus of this session*
- Building Python extensions in C/C++
  - *in the appendix*

Resources

- Main site:
  - *www.python.org*
  - *Download binaries, sources, documentation*
  - *Contributed packages*

- Books:
  - *“Programming Python” by Mark Lutz*
  - *“Learning Python” by Mark Lutz*
  - *Lots of others on more specific topics*

Overview of the Python Language

- Built-in objects and their operators
  - *Numbers, strings, lists, dictionaries, tuples*
  - *Files*
  - *Object properties*
- Statements
  - *Assignment, expressions,* **`print`**, **`if`**, **`while`**
  - **`break`**, **`continue`**, **`pass`**, *loop* **`else`**
  - **`for`**
- Functions
  - *Scope rules*
  - *Argument passing*
  - *Callable objects*
- Modules and Packages
  - *Name qualification*
  - **`import`**
  - *Scope objects*
- Classes
  - *Declarations and definitions*
  - *Inheritance*
  - *Overloading operators*
- Exceptions
  - *Raising and catching*
  - *Exception hierarchies*

Built-in objects

- Preview

Type	*Sample*
Numbers	1234, 3.1415, 999L, 3+4j
Strings	‘help’, “hello”, “It’s mine”
Lists	[‘this’, [‘and’, 0], 2]
Dictionaries	{‘first’: ‘Jim’, ‘last’: ‘Brown’}
Tuples	(1, “this”, ‘other’)
Files	open(‘sample.txt’, ‘r’)

Operators and precedence

Operators	*Description*
or, lambda	Logical ‘or’, anonymous function
and	Logical ‘and’
<, <=, >, >=, ==, <>, != is, is not, in, not in	Comparisons, sequence membership
x | y	Bit-wise ‘or’
x ^ y	Bit-wise ‘exclusive or’
x & y	Bit-wise ‘and’
x << y, x >> y	Shift left and right
+, -	Addition, subtraction
*, /, %	Multiplication/repetition, division, remainder/format
-x, +x, ~x	Unary minus, plus, and compliment
x[i], x[I:j], x.y, x(…)	Indexing, slicing, qualification, function call
(…), […], {…}, `…`	Tuple, list, dictionary, conversion to string

Numbers

- Expressions
  - *The usual operators*
  - *Bit-wise operators (same as C)*
  - *Change precedence and association using parentheses*
  - *In expressions with mixed types, Python coverts upwards*
- Numeric constants

Constant	*Meaning*
1234, -1234	integers (C long)
9999L	arbitrary precision integers
3.1415, 6.023e-23	floats (C doubles)
0177, 0xdeadbeef	octal and hex constants
j, 1.0 – 3.14j	complex numbers

Strings

- Immutable ordered sequences of characters
  - *no char type: 'a' is an one-character string*
- Constants, operators, utility modules
- Common operations

Operation	*Meaning*
s = ''	empty string
s = "It's mine"	double quotes
s = """ … """	triple quote blocks
s1 + s2, s1 * 4	concatenate, repeat
s[i], s[i:j], len(s)	index, slice, length
for x in s, 'm' in s	iteration, membership

Lists

- Mutable ordered sequences of object references
  - *variable length, heterogeneous, arbitrarily nestable*
- Common list operations

Operation	*Meaning*
L = []	empty list
L = [1, 2, 3, 4]	Four items, indexes: 0..3
[‘one’, [‘two’, ‘three’], ‘four’]	nested lists
L[j], L[j:k], len(L)	index, slice, length
L1 + L2, L*3	concatenate, repeat
L.sort(), L.append(4)	object methods
del L[k], L[j:k] = []	shrink
L[j:k] = [1, 2, 3]	slice assignment
range(4), xrange(0,5)	create integer lists
for x in L, 1 in L	iteration, membership

Dictionaries

- Mutable unordered binary associations (maps)
  - *accessible by key*
- Common operations

Operation	*Meaning*
d = {}	empty dictionary
d1 = {‘last’: ‘Brown’, ‘height’: 1.85}	Two items
d2 = {‘person’ : {‘last’: ‘Brown’, ‘height’: 1.85}, ‘state’: ‘dead’}	nested dictionaries
d1[‘last’], d2[‘person’][‘last’]	indexing
d.has_key(‘last’)	method
len(d)	number of entries
del d[key], d[key] = value	remove, add/change

Tuples

- Immutable ordered sequences of object references
- Common operations:

Operation	*Meaning*
t = ()	empty tuple
t = (1,)	tuple with one item
t = (1, 2, 3, 4)	tuple with four items
t = (1, 2, (3, 4), 5)	nested tuples
t1 + t2, t * 4	concatenate, repeat
t[i], t[i:j], len(t)	index, slice, length
for x in t, i in t	iteration, membership

Files

- A Python object wrapped around the C stdio system
  - *an extension type, written in C*
- Common operations

Operation	*Meaning*
out = open(“hello.txt”, “w”)	create output file
in = open(“hello.txt”, “r”)	create input file
in.read(), in.read(1)	read file, read byte
in.readline(), in.readlines()	read a line, fill a list of strings
out.write(s), out.writelines(L)	write a string, write a list of strings
out.close()	flush and close file explicitly

Built-in objects - summary

- Everything is an object – PyObject
- Assignments create new *references* to *existing* objects
- Containers can hold any kind of object
- Changing a mutable object affects all references
- Objects belong to categories with common operations

Object Type	*Category*	*Mutable?*
Numbers	Numeric	No
Strings	Sequence	No
Lists	Sequence	Yes
Dictionaries	Mapping	Yes
Tuples	Sequence	No
Files	Extension	Yes

String coercions and the format operator

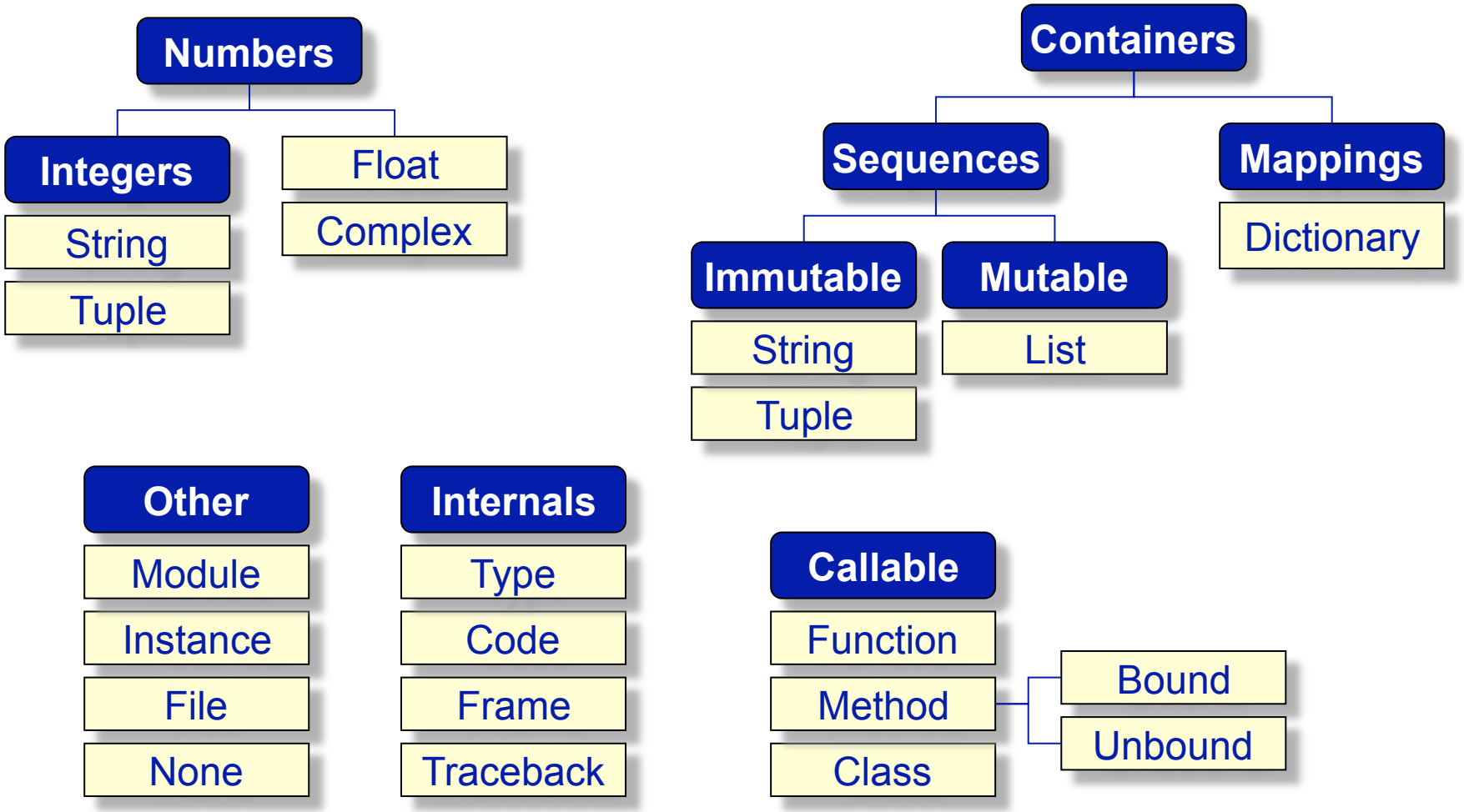
- All objects can potentially be represented as strings
- Coersion can be triggered explicitly
  - *using the `` ` ` `` operator*
  - *using the* **`repr()`** *built-in function*
  - *using the* **`str()`** *built-in function*
- The string format operator '%'
  - *the python equivalent of* **`sprintf`**
  - *binary operator:*
    - *LHS is a string that may include format specifications*
    - *RHS is a tuple of the arguments that replace the format specifications*
  - *accepts the same format specifications as* **`printf`**
- Example:

```
filename = "%s-%05d.dat" % (hostname, pid)
```

Truth and equality

- The following values are considered “false”
  - *The special object* `None`
  - *The number* `0`
  - *Any empty container:* `""`, `[]`, `{}`, `()`
- All other values are “true”
- Operators
  - *Identity:* `is`
  - *Membership:* `in`
  - *The usual relational operators – borrowed from C*
- Object comparisons
  - *Strings are compared lexicographically*
  - *Nested data structures are checked recursively*
  - *Lists and tuples are compared depth first, left to right*
  - *Dictionaries are compared as sorted (key, value) tuples*
  - *User defined types can specify comparison functions using overloaded operators*

Python type hierarchy



Python syntax

- Comments: from a '#' to the end of the line
- Indentation denotes scope
  - *avoid using tabs*
- Statements end at the end of line, or at ';'
  - *open delimiter pairs imply continuation*
  - *explicit continuation with '\' but considered obsolete*
- Variable names
  - *underscore or letter, followed by any number of letters, digits and underscores*
  - *case sensitive*
    - *Guido wanted to take this away, but wisdom prevailed…*

Reserved words

access	and	break	class
continue	def	del	elif
else	except	exec	finally
for	from	global	if
import	in	is	lambda
not	or	pass	print
raise	return	try	while

Printing expressions

- The statement **`print`**
  - *converts objects to string*
  - *and writes the string to the* **`stdout`** *stream*
- Adds a line-feed
  - *to suppress, add a trailing comma*

```
print <expression>
print <expression>,
```

- The obligatory “Hello world” program:

```
print “Hello world”
```

Assignments

- Explicitly, using **=**

```
<name> = <expression>
```

- Implicitly, using **import**, **def**, **class**

```
import <module>
from <module> import <name>
from <module> import <name> as <alias>

def <name>(<parameter list>):

class <name>(<ancestor list>):
```

Selections

- Using **`if`**

```
if <expression>:
    <statements>
elif <expression>:
    <statements>
else:
    <statements>
```

- No **`switch`** statement
  - use **`if`**
  - or lists and dictionaries

Explicit loops

- **`while`**

```
while <expression>:
    <statements>
else:
    <statements>
```

- **`for`**

```
for <name> in <container>:
    <statements>
else:
    <statements>
```

- The **`else`** part is optional
  - it is executed when exiting the loop normally
- Other relevant statements: **`break`**, **`continue`**, **`pass`**

Function basics

- General form:

```
def <name>(<parameter list>):
    <statements>
    return <expression>
```

- Creates a function object and assigns it to the given name
  - return sends an object to the caller (optional)
  - arguments passed "*by assignment*"
  - no declarations of arguments, return types and local variables
- Example:

```
def isServer(processor_id):
    if processor_id is 0: return 1
    return 0
```

Function scoping rules

- Enclosing module acts as the global scope
- Each call to a function creates a new local scope
- All assignments in the function body are local
  - *unless declared global*
- All other names used should be global or built-in
  - *references search three name scopes: local, global, built-in*

```
root_id = 12

def isServer(processor_id):
    if processor_id is root_id: return 1
    return 0

def setServer(processor_id):
    global root_id
    root_id = processor_id
    return
```

Function arguments

- Passing rules:
  - *Arguments are passed by creating a local reference to an existing object*
  - *Re-assigning the local variable does not affect the caller*
  - *Modifying a mutable object through the local reference impacts caller*
- Argument matching modes:
  - *by position*
  - *by keyword*
  - *using varargs:*
    - **: places non-keyword arguments in a tuple*
    - ***: places keyword arguments in a dictionary*
  - *using default values supplied in the function declaration*
- Ordering rules:
  - *declaration: normal, *arguments, **arguments*
  - *caller: non-keyword arguments first, then keyword*

Matching algorithm

- Assign non-keyword arguments by position
- Assign keyword arguments by matching names
- Assign left over non-keyword arguments to ***name** tuple
- Assign extra keyword arguments to ****name** dictionary
- Unassigned arguments in declaration get their default values

Functions as objects

- Function objects can be assigned, passed as arguments, etc.

```
import string

def convert(string, conversion=string.lower):
    return conversion(string)

greeting = " Hello world! "
operation = string.strip
convert(greeting, operation)
```

- Nameless function objects can be created using **`lambda`**

```
greeting = " Hello world! "
operation = lambda x: x[1:-1]
operation(greeting)
```

Namespaces

- Modules are created by
  - *statically linking code with the interpreter executable*
  - *interpreting Python files*
    - *source -> byte compiled on first import*
  - *dynamically loading shared objects during interpretation*
    - *extensibility*
- Packages are directories with modules
  - *That contain a special file* **`__init__.py`**
    - *Whose attributes affect how* **`import`** *works*
  - *The directory name becomes the package name*
- The search path for modules is controlled
  - *At interpreter compile time*
  - *By the interpreter's current working directory*
  - *By reading user defined settings*
    - *e.g. $PYTHONPATH or the win32 registry*
  - *By modifying* `sys.path`

Access to namespaces

- Modules are namespaces
  - *They introduce a scope*
  - *Statements run on **first** import*
  - *All top level assignments create module attributes*
- Packages are namespaces
  - *They introduce a scope*
  - *Their attributes are set by interpreting the special file __init__.py on first import*
- Names are accessed with the **`import`** implicit assignment statement

```
import <namespace>
from <namespace> import <name>
from <namespace> import *
from <namespace> import <name> as <alias>
```

- Name qualifications allow fine tuning of the list of imported symbols

```
from pyre.support.debug import DebugCenter
```

Namespaces as objects

- Modules and packages are objects:

```
def load(material):
    exec "from pyre.materials import %s as model" % material
    return model

materialModel = load("perfectGas")

material = materialModel.newMaterial(options)
```

- Dynamic programming!

Classes

- Classes are *object factories:*
  - Construct new objects with state and behavior
  - Using a class name as a function creates calls the constructor
  - Each instance inherits all class attributes
  - Assignments in class statements create class attributes
  - Assignments to **`self`** create per-instance attributes

```
class Body:

    _type = "Generic body"

    def type(self): return self._type

    def __init__(self):
        self._rep = None
        return
```

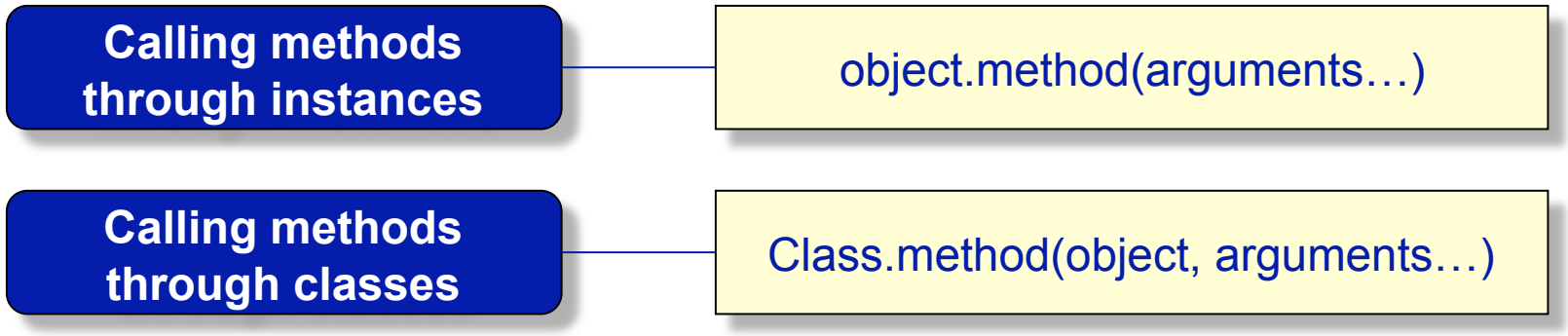
Inheritance

- Specialization through inheritance
  - *Superclasses must be listed during the class declaration*
  - *Classes inherit attributes from their ancestors*
  - *Instances inherit attributes from all accessible classes*

```
class Cylinder(Body):

    _type = "Cylinder"

    # def type(self): return self._type
    def radius(self): return self._radius
    def height(self): return self._height

    def __init__(self, radius, height):
        Body.__init__(self)
        self._radius = radius
        self._height = height
        return
```

Methods

- The class statement creates and assigns a class object
- Calling class objects as functions creates instances
- Class methods provide behavior for the instance objects
- Methods are nested functions with at least one parameter
  - *That receives the instance reference*
  - *Named* `self` *by convention*
- Methods are public and virtual
- Syntax:



Class glossary

- Class
  - *A blueprint for the construction of new types of objects*
- Instance of a class
  - *An object created from a class constructor*
- Member
  - *An attribute of an instance that is bound to an object*
- Method
  - *An attribute of a class instance that is bound to a function object*
- Self
  - *The conventional name given to the implied instance object in methods*

Overloading operators in classes

- **Don't**
- Classes can intercept normal Python operations
- All Python expressions can be overloaded
- Special method names.
- Examples:

Method	*Overloads*
__init__	Construction: x = X()
__del__	Destruction
__repr__	Representation: `x`, repr(x)
__str__	String coercion: str(x)
__len__	Size, truth tests: len(x)
__cmp__	Comparisons: x < object
__call__	Function calls: x()

Method	*Overloads*
__getattr__	Qualification: x.undefined
__getitem__	Indexing: x[5]
__setitem__	Indexing: x[5] = 0
__add__	Addition: x + other
__radd__	Addition: other + x
__and__	Logic: x and object
__or__	Logic: x or object

Namespace rules

- The complete story
  - *Unqualified names are looked up in the three default lexical namespaces*
  - *Qualified names conduct a search in the indicate namespace*
  - *Scopes initialize object namespaces: packages, modules, classes, instances*
- Unqualified names, e.g. **`name`**,
  - *Are global on read*
  - *Are local on write, unless declared global*
- Qualified names, e.g. **`object.name`**,
  - *Are looked up in the indicated namespace*
    - *Module and package*
    - *Instance, class and ancestors (depth first, left to right)*
  - *References and assignments modify the qualified attributes*
- Namespace dictionaries:
  - **`__dict__`**
  - *Name qualification is identical to dictionary lookup!*

Classes as objects

- Classes are objects
- Examples:

```
def newSolver():
    from Adlib import Adlib
    return Adlib

solver = newSolver()
```

```
def load(material):
    exec "from %s import %s as factory" % (material, material)
    return factory

materialModel = load("perfectGas")(options)
```

Methods as objects

- Unbound method objects

```
method = Object.method
object = Object()

method(object, arguments)
```

- Bound method objects

```
object = Object()
method = object.method

method(arguments)
```

Exceptions

- A high level control flow device
  - *non-local*
- Exceptions are used to signal
  - *critical errors*
  - *but also recoverable runtime failures*
- Exceptions are raised by the interpreter
  - *there is an extensive exception class hierarchy*
- The statement **`raise`** triggers an exception
- The statement **`try`** sets up a net for catching them
- Should be treated as seriously as any other part of the application
  - *exception class hierarchies*

Raising exceptions

- Exceptions are triggered by **`raise`**

```
raise <string>
raise <string>, <data>
raise <class>, <instance>
raise <class instance>
```

- The last two are identical
  - *the preferred way to raise exceptions*
- Exceptions used to be strings
  - *obsolete*
  - *because it is a bad practice*

Catching exceptions

- Basic forms:

```
try:
    <statements>
except <class>:
    <statements>
except <class>, <data>
    <statements>
else:
    <statements>
```

```
try:
    <statements>
finally:
    <statements>
```

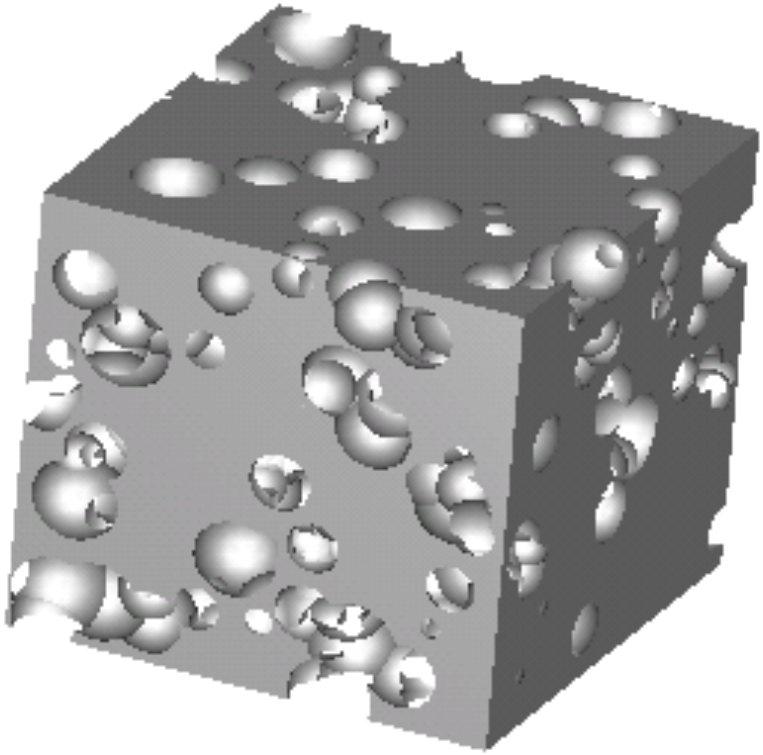
Object oriented programming

- No mathematically precise/sharp definition
- Application designs expressed as “objects” and their collaborations
- Characteristics:
  - *class: a user defined **type** that captures the essence of a part of the model*
  - *object: an instance of a class*
  - *method: an ability (function) of an object*
  - *attribute: a piece of storage (field) assigned to an object*
- Mechanisms
  - *message passing: calling a method*
  - *inheritance: a mechanism for expressing relationships between classes*
  - *polymorphism: derived class specific implementation of an ancestor’s method*
- Design goals
  - *encapsulation: hide the implementation details; focus on the interface*
  - *abstraction: model systems by focusing on the relevant irreducible concepts*

Case study: CSG

- Goal: design a set of classes to encapsulate the construction of solids using
  - *Solid primitives:*
    - *Block, Sphere, Cylinder, Cone, Pyramid, Prism, Torus*
  - *Basic operations:*
    - *Unions, intersections, differences*
    - *Rotations, reflections, translations, dilations*
    - *Reversals*
- Scope: delegate actual body construction to a CSG package
  - *Only record the solid primitives and operations required in the construction*
  - *Read and write solids to (XML) files*
  - *Extension module: interface with an actual CSG package*
    - *ACIS, a C++ class library*
    - *Carry out the construction, surface meshing for viewing, etc.*

Examples



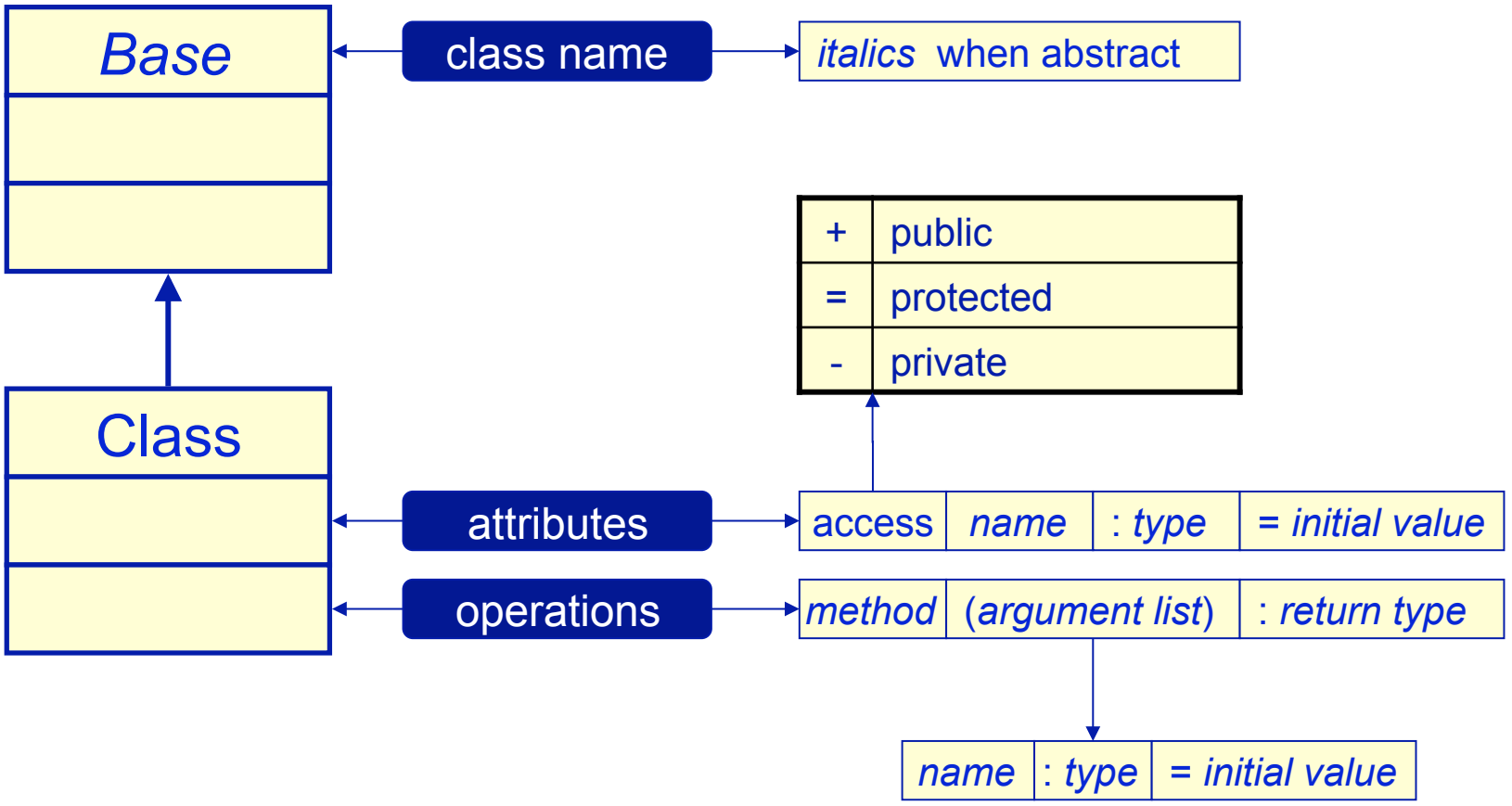
Motivation

- Object oriented skills
  - *Class hierarchy design*
  - *Introduction to UML*
  - *Introduction to Design Patterns*
- Python skills
  - *Class hierarchy implementation*
  - *Package design*
  - *Interaction with a C++ class library*
    - *Design and implementation of an extension module*
- Raising the bar
  - *Reading and writing XML files*
  - *Graphical browser*
  - *Visualization*

Implementation

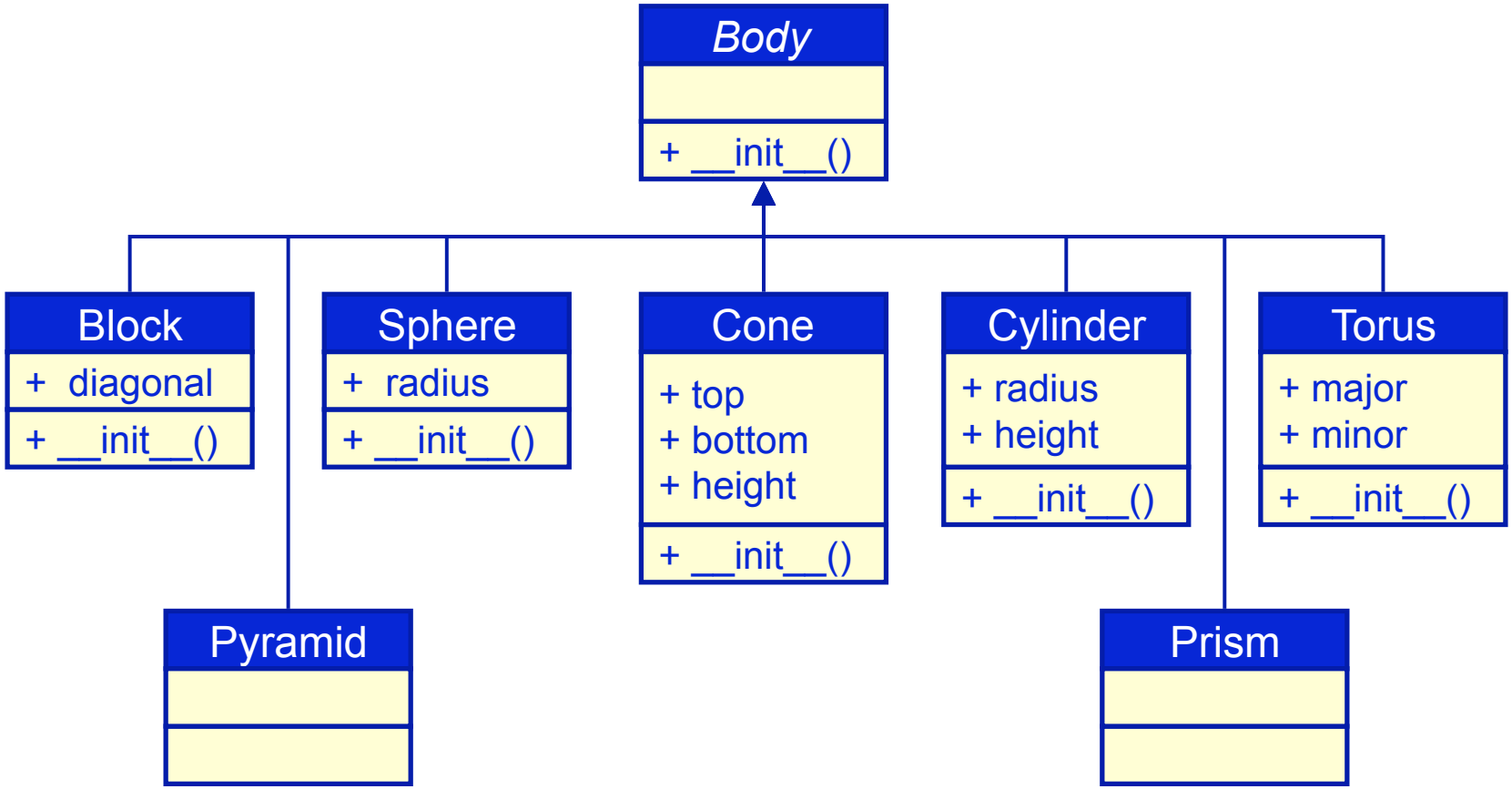
- Use Python to express the solution
  - *To take advantage of:*
    - *Rapid development*
      - *Short development cycle, expressiveness, loose typing*
    - *Access to large set of library packages*
      - *E.g. GUI toolkits, regular expressions*
  - *But carefully:*
    - *Proper project structure*
    - *Should not abuse the lack of strong typing*
    - *Proper error handling*
    - *Might have to migrate parts to C++*

UML class diagrams



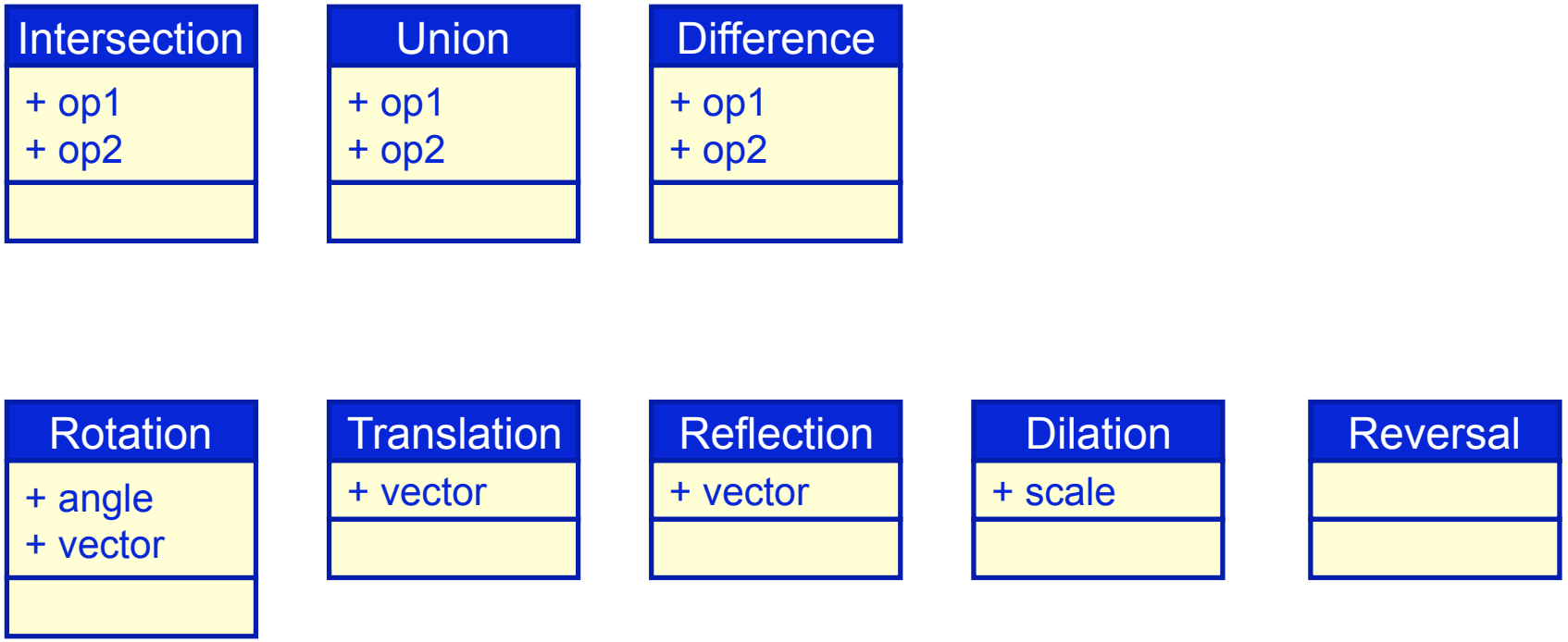
Solid primitives

- Here is a possible class hierarchy



Operations

- What is the best way to represent these?



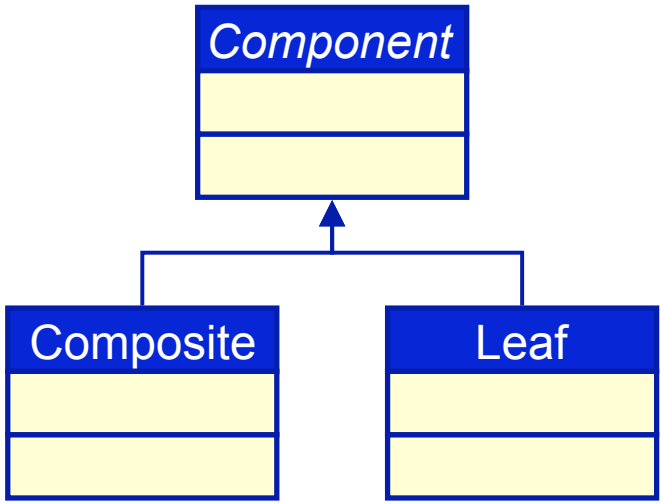
Design patterns

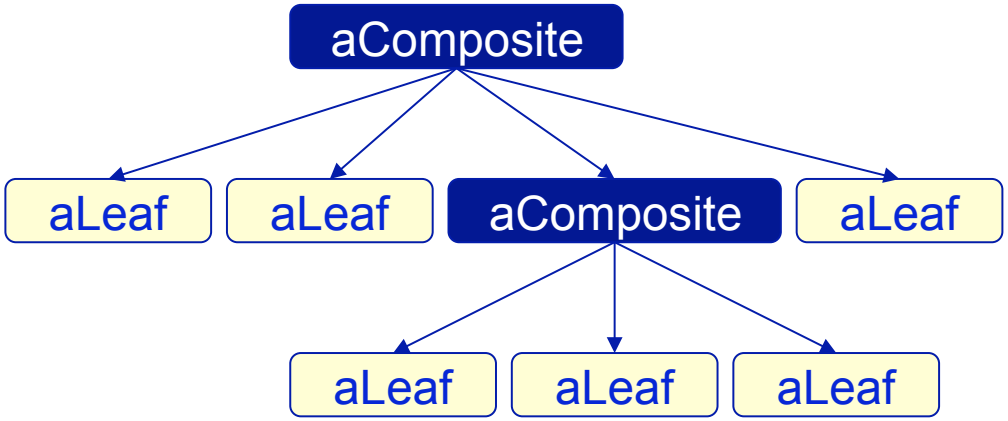
"A design pattern is a description of a set communicating classes and objects that are customized to solve a general design problem in a particular context"

- Introduction:
  - *"Design Patterns: Elements of Reusable Object-Oriented Software"*
    - *by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides*
    - *Addison-Wesley, 1995, QA76.64.D47*
- Patterns have four parts:
  - *name*
    - *increases the design vocabulary*
  - *problem*
    - *a description of the problem and its context*
  - *solution*
    - *consists of classes and their specific collaborations*
  - *consequences*
    - *design and implementation trade-offs*

Composite

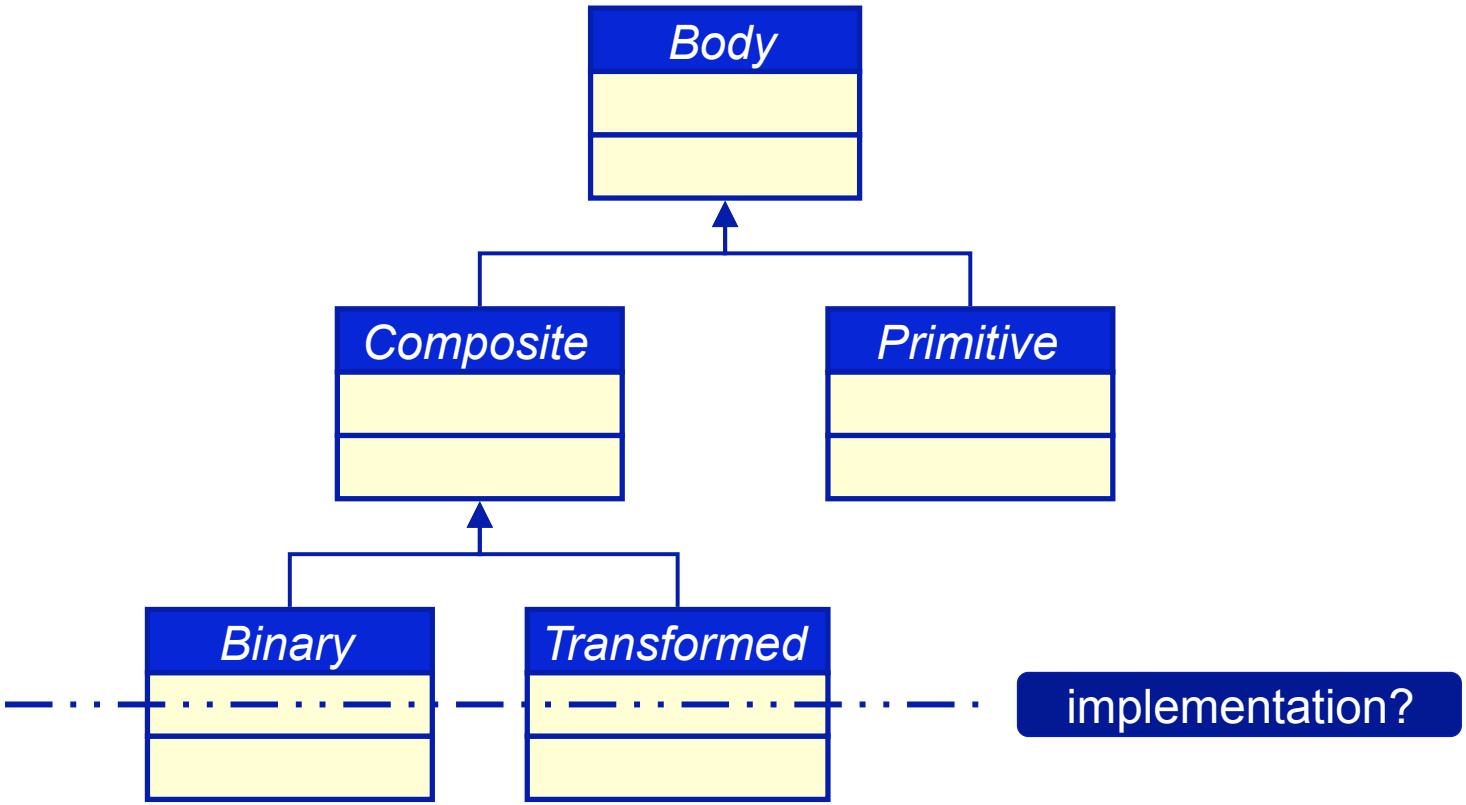
- Intent:
  - *"Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly"*



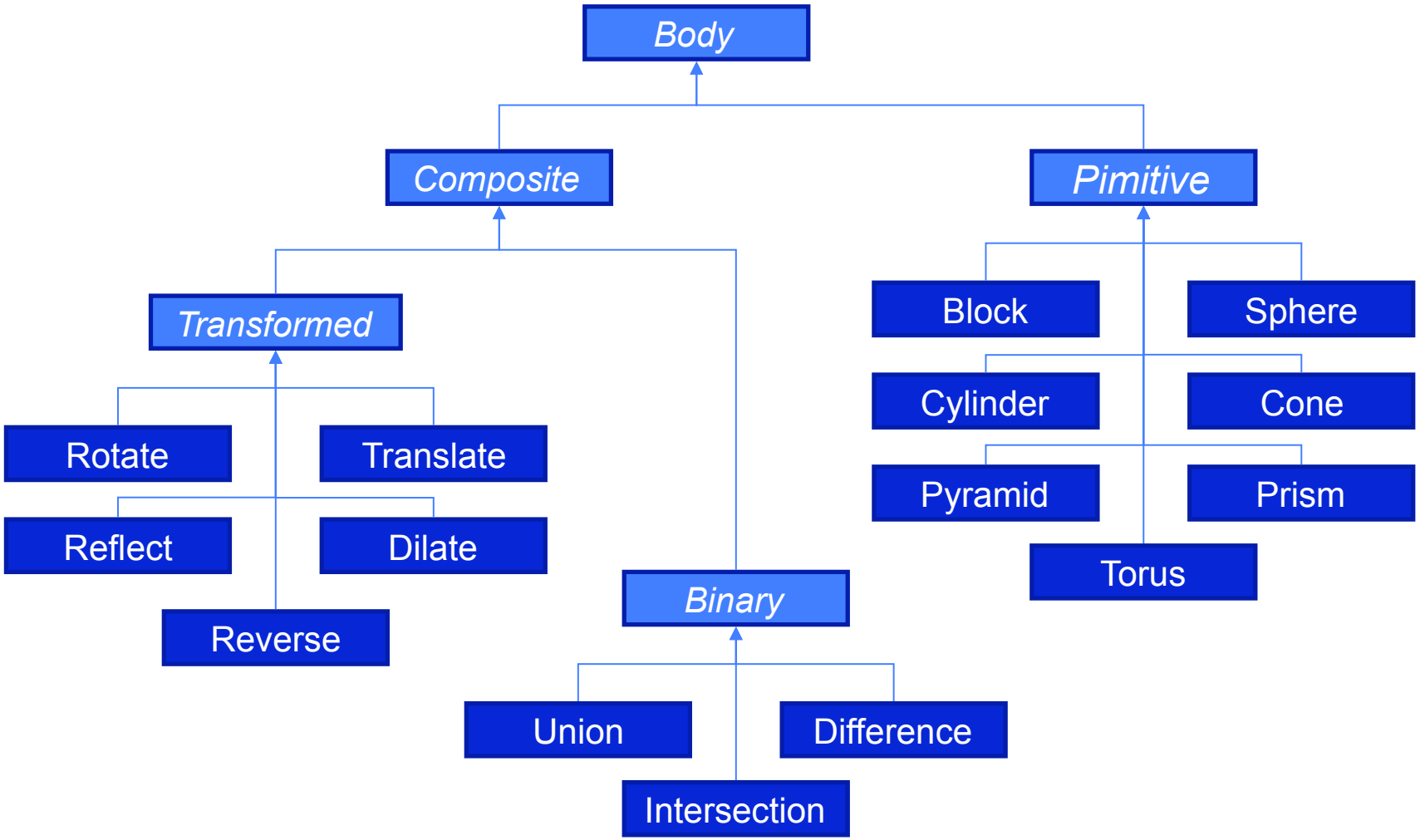


Suggested solution

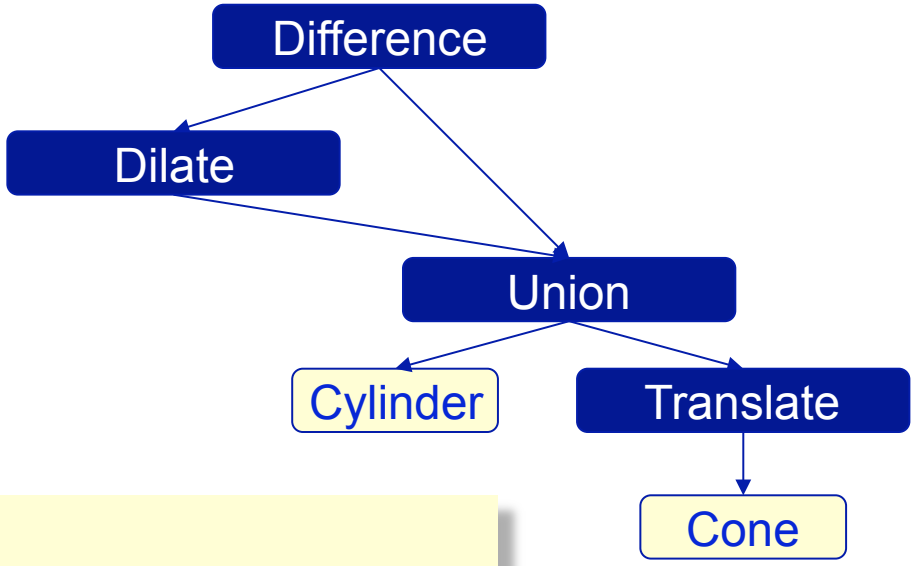
- A hierarchy of abstract base classes:



The class hierarchy



Putting it all together




```
body = Cylinder(radius, height)
cone = Cone(radius/2, radius, height)
cap = Translate(cone, (0, 0, height/2))

innerWall = Union(body, cap)
outerWall = Dilate(innerWall, scale=1.1)

shell = Difference(outerWall, innerWall)
```

Checkpoint – assessing the design

- Encapsulation
- Completeness
  - *We appear to have captured the entire set of primitive objects*
    - *the hierarchy is not very likely to change*
- Capabilities
  - *Not much more than "byte classification and storage"*
    - *solved the data-centric part of the problem only*
  - *What can we do with our Composite?*

Adding operations

- Examples
  - *Realizing bodies by using solid modeling engines*
  - *Cloning*
  - *Writing bodies to files*
    - *using a variety of output formats*
- Others?
- Operations on body hierarchies require tree traversals that
  - *are polymorphic on the type of node being traversed*
  - *can maintain traversal state relevant for the type of operation*

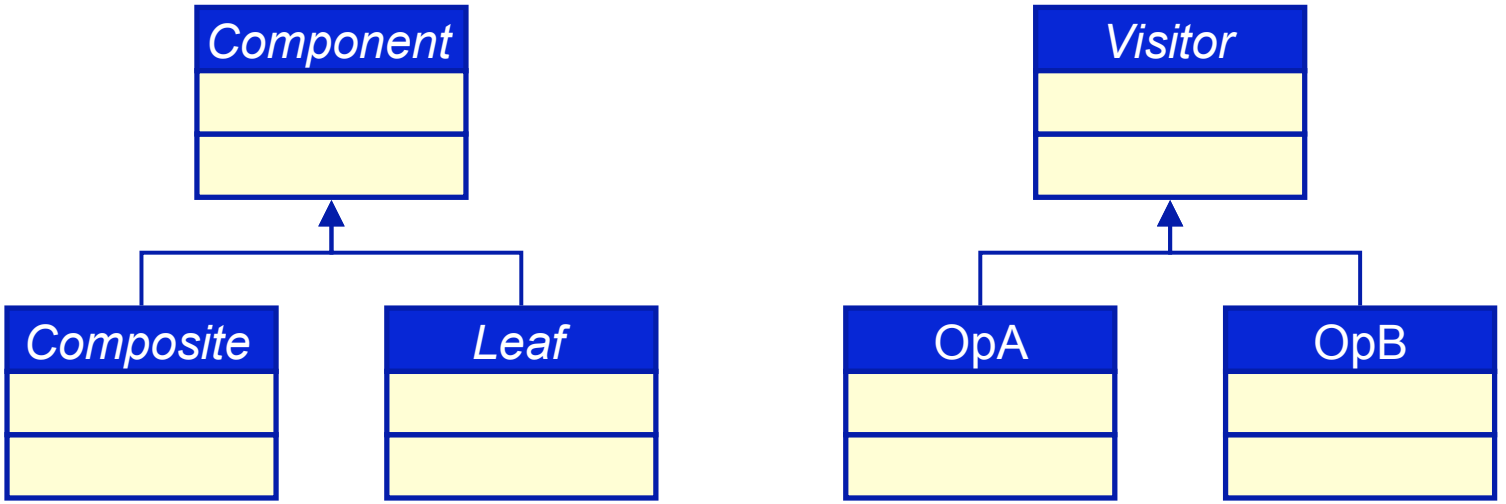
Implementation strategies

- Add (virtual) functions for each operation
  - *requires extensive updates for each new operation*
- Use run-time type information

- Double dispatch

Visitor

- Intent
  - *"Represent the operations to be performed on the elements of an object structure. Visitor lets you defines a new operation without changing the classes of the elements on which it operates"*



Visitor implementation

- Augment Composite by giving each class a type identification method. For example, Cylinder gets:

```
def identify(self, visitor):
    return visitor.onCylinder(self)
```

- Visitor descendants implement the type handler:

```
def onCylinder(self, cylinder):
    # do cylinder specific things

    return
```

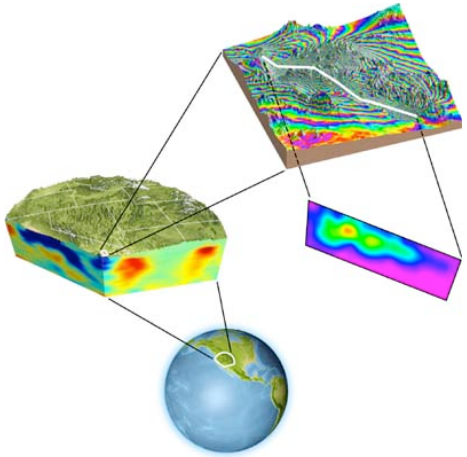
- Best when the Composite hierarchy is stable

Building a distributed component framework

- Putting OO to work!

- What is a distributed component framework?
  - *what is a component?*
  - *what is a framework?*
  - *why be distributed?*
- Why bother building a framework?
  - *is it the solution to any **relevant** problem?*
  - *is it the right solution?*
- High level description of the specific solution provided by ***pyre***

Pyre overview

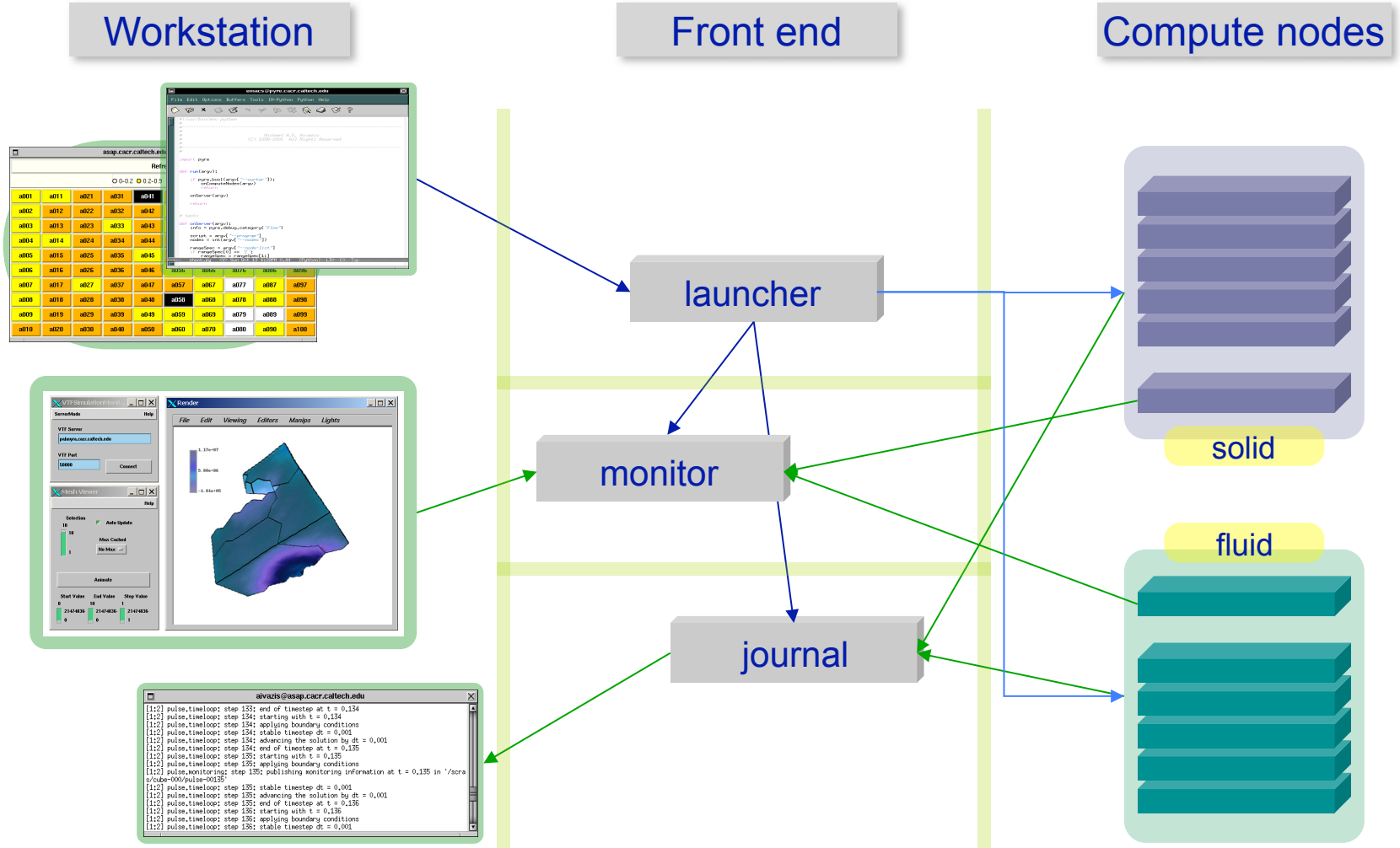


- Projects
  - *Caltech ASC Center (DOE)*
  - *Computational Infrastructure in Geodynamics (NSF):*
  - *DANSE (NSF)*
- Portability:
  - *languages: C, C++, F77, F90*
  - *compilers: all native compilers on supported platforms, gcc, Absoft, PGI*
  - *platforms: all common Unix variants, OSX, Windows*
- Statistics:
  - *1200 classes, 75,000 lines of Python, 30,000 lines of C++*
  - *Largest run: **nirvana** at LANL, 1764 processors for 24 hrs, generated 1.5 Tb*

User stereotypes

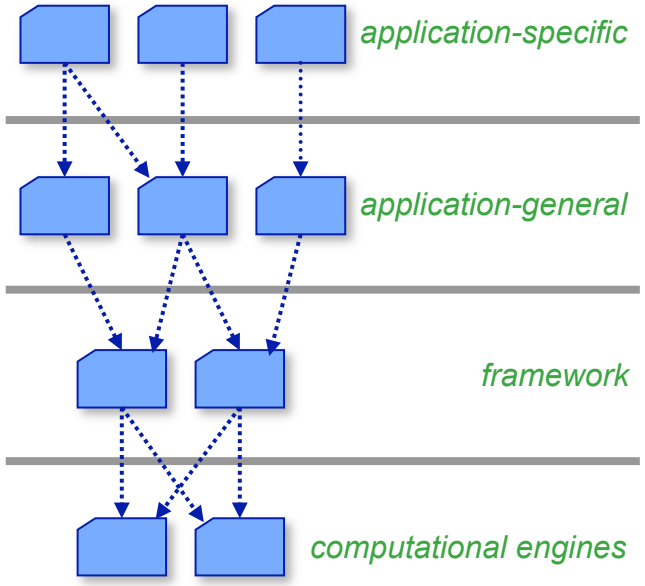
- End-user
  - *occasional user of prepackaged and specialized analysis tools*
- Application author
  - *author of prepackaged specialized tools*
- Expert user
  - *prospective author/reviewer of PRL paper*
- Domain expert
  - *author of analysis, modeling or simulation software*
- Software integrator
  - *responsible for extending software with new technology*
- Framework maintainer
  - *responsible for maintaining and extending the infrastructure*

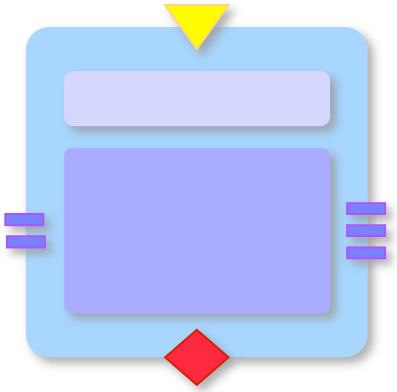
Distributed services



Pyre: the integration architecture

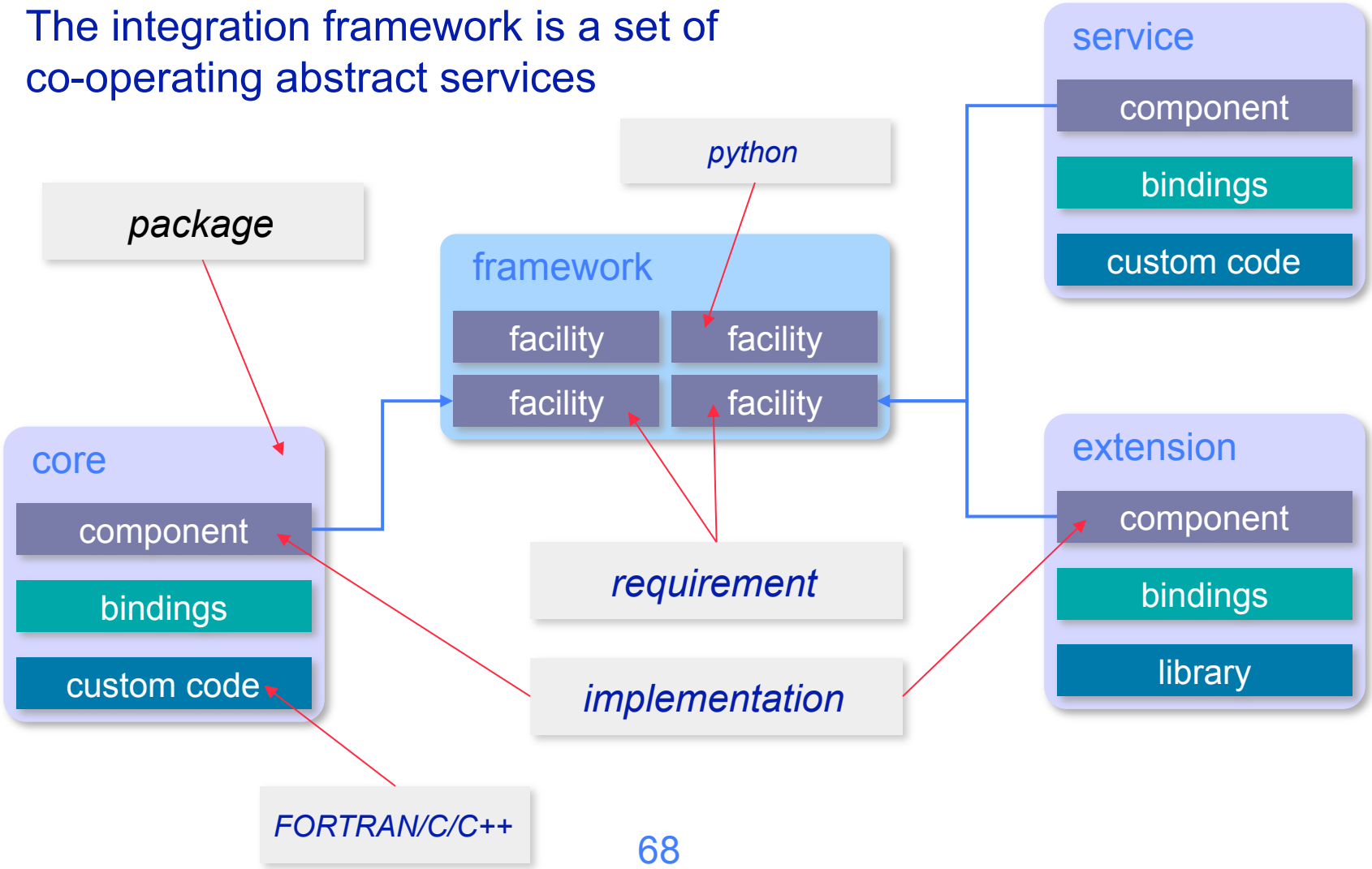
- Pyre is a *software architecture*:
  - *a specification of the organization of the software system*
  - *a description of the crucial structural elements and their interfaces*
  - *a specification for the possible collaborations of these elements*
  - *a strategy for the composition of structural and behavioral elements*
- Pyre is multi-layered
  - *flexibility*
  - *complexity management*
  - *robustness under evolutionary pressures*
- Pyre is a *component framework*



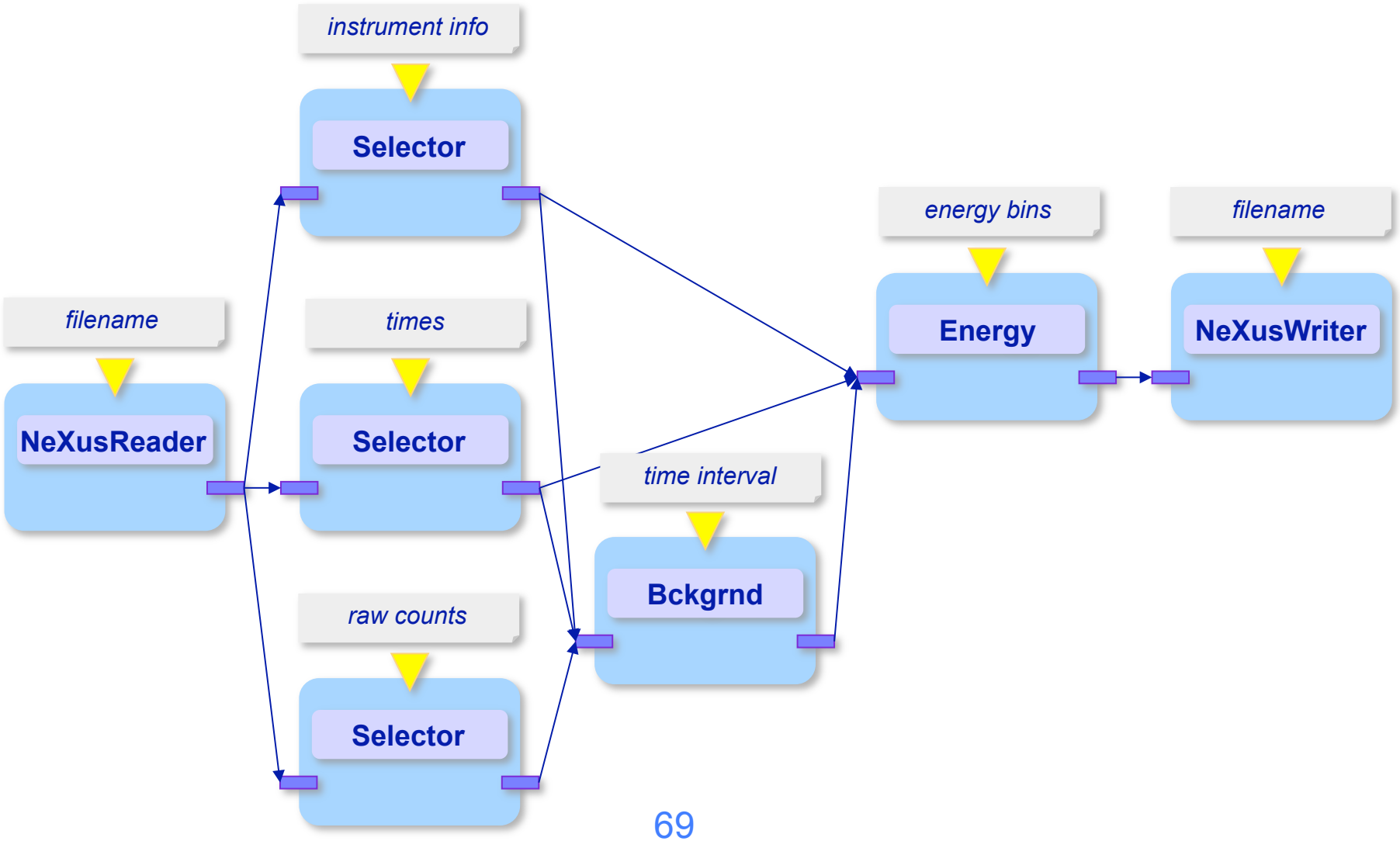


Component architecture

- The integration framework is a set of co-operating abstract services



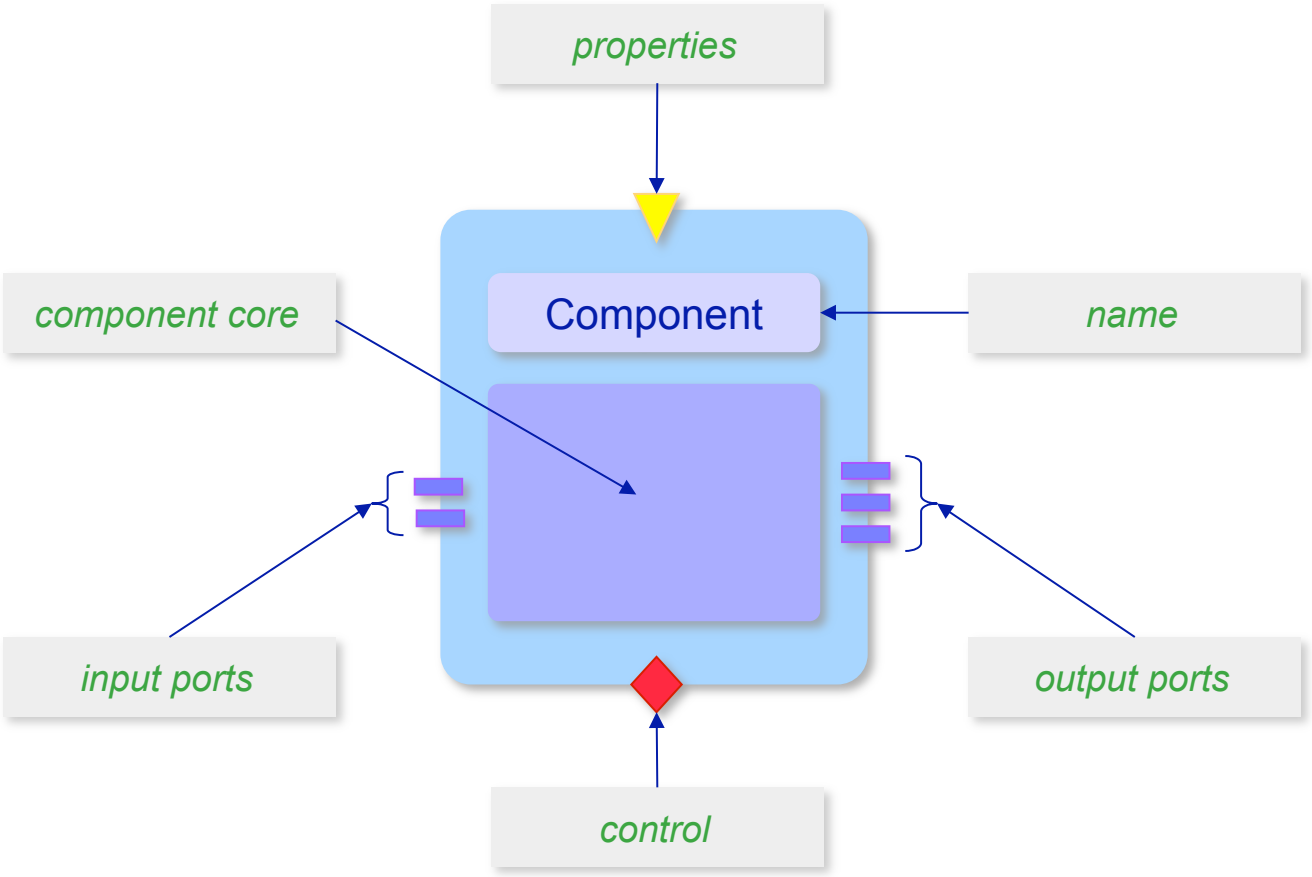
Example application



Encapsulating critical technologies

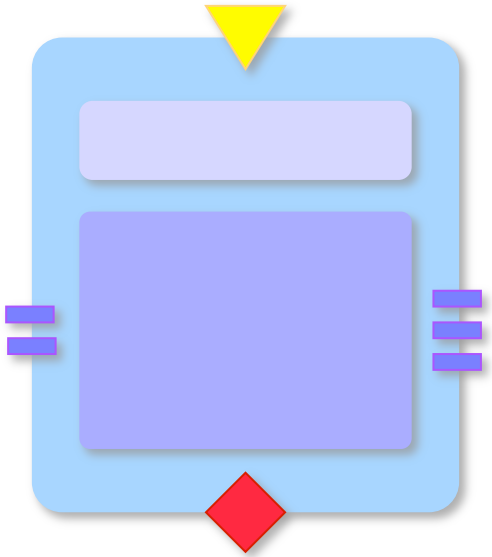
- Extensibility
  - *new algorithms and analysis engines*
  - *technologies and infrastructure*
- High-end computations
  - *visualization*
  - *easy access to large data sets*
    - *single runs, backgrounds, archived data*
    - *metadata*
  - *distributed computing*
  - *parallel computing*
- Flexibility:
  - *interactivity: web, GUI, scripts*
  - *must be able to do almost everything on a laptop*

Component schematic



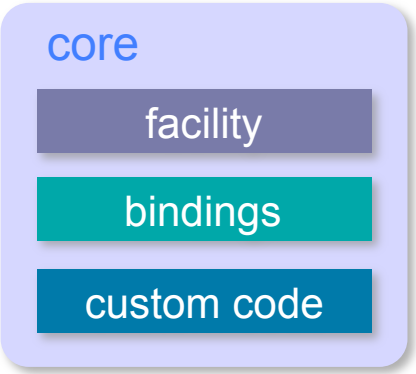
Component anatomy

- Core: encapsulation of computational engines
  - *middleware that manages the interaction between the framework and codes written in low level languages*
- Harness: an intermediary between a component's core and the external world
  - *framework services:*
    - *control*
    - *port deployment*
  - *core services:*
    - *deployment*
    - *launching*
    - *teardown*



Component core

- Three tier encapsulation of access to computational engines
  - *engine*
  - *bindings*
  - *facility implementation by extending abstract framework services*
- Cores enable the lowest integration level available
  - *suitable for integrating large codes that interact with one another by exchanging complex data structures*
  - *UI: text editor*

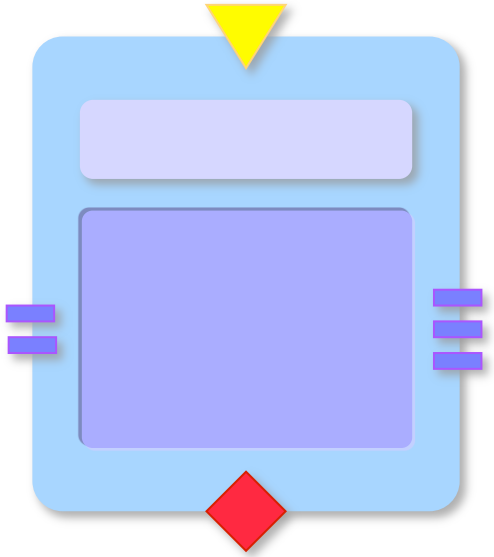


Computational engines

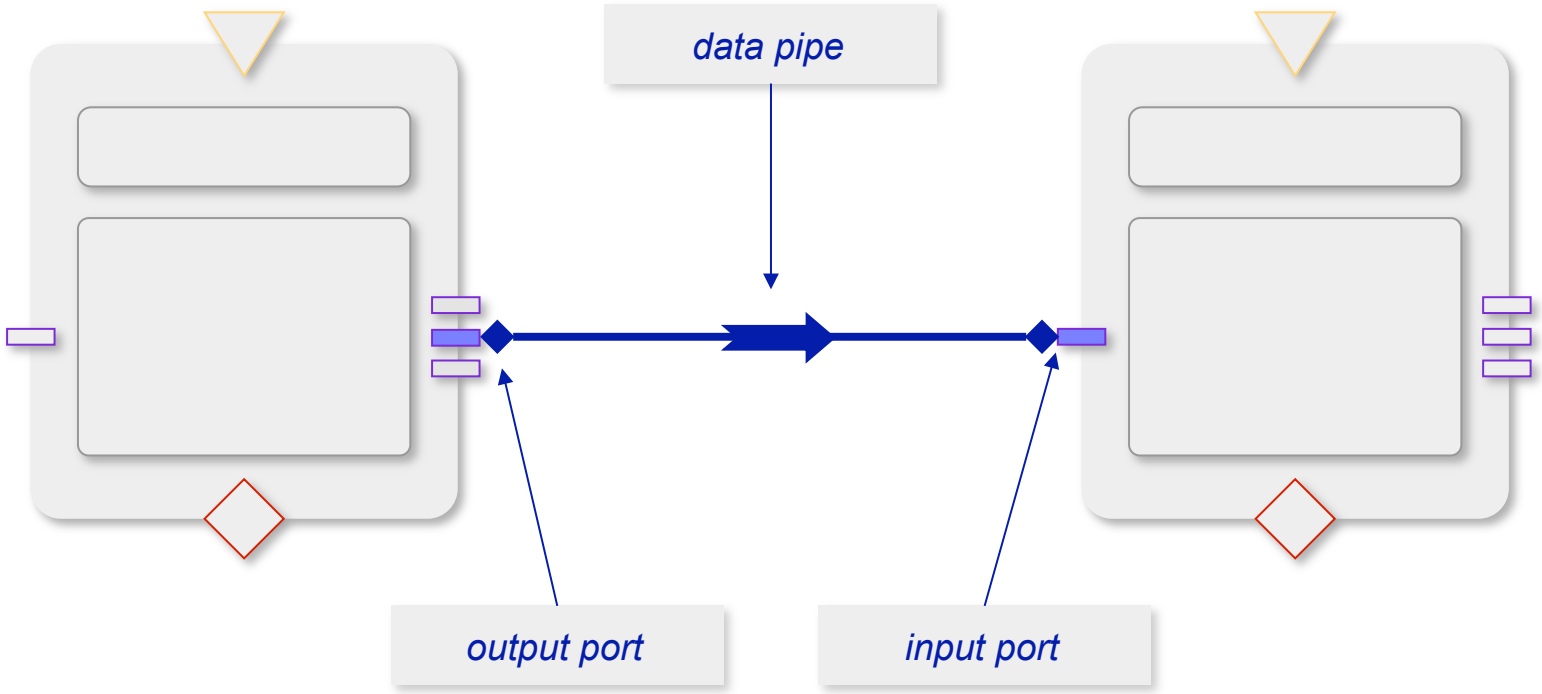
- Normal engine life cycle:
  - *deployment*
    - *staging, instantiation, static initialization, dynamic initialization, resource allocation*
  - *launching*
    - *input delivery, execution control, hauling of output*
  - *teardown*
    - *resource de-allocation, archiving, execution statistics*
- Exceptional events
  - *core dumps, resource allocation failures*
  - *diagnostics: errors, warnings, informational messages*
  - *monitoring: debugging information, self consistency checks*
- Distributed computing
- Parallel processing

Component harness

- The harness
  - *collects and delivers user configurable parameters*
  - *interacts with the data transport mechanisms*
  - *guides the core through the various stages of its lifecycle*
  - *provides monitoring services*
- Parallelism and distributed computing are achieved by specialized harness implementations
- The harness enables the second level of integration
  - *adding constraints makes code interaction more predictable*
  - *provides complete support for an application generic interface*



Data transport



Ports and pipes

- Ports further enable the physical decoupling of components by encapsulating data exchange
- Runtime connectivity implies a two stage negotiation process
  - *when the connection is first established, the io ports exchange abstract descriptions of their requirements*
  - *appropriate encoding and decoding takes place during data flow*
- Pipes are data transport mechanisms chosen for efficiency
  - *intra-process or inter-process*
  - *components need not be aware of the location of their neighbors*
- Standardized data types obviate the need for a complicated runtime typing system
  - *meta-data in a format that is easy to parse (XML)*
  - *tables*
  - *histograms*

Component implementation strategy

- Write engine
  - *custom code, third party libraries*
  - *modularize by providing explicit support for life cycle management*
  - *implement handling of exceptional events*
- Construct python bindings
  - *select entry points to expose*
- Integrate into framework
  - *construct object oriented veneer*
  - *extend and leverage framework services*
- Cast as a component
  - *provide object that implements component interface*
  - *describe user configurable parameters*
  - *provide meta data that specify the IO port characteristics*
  - *code custom conversions from standard data streams into lower level data structures*

- All steps are well localized!

Writing python bindings

- Given a “low level” routine, such as

```
double stableTimeStep(const char *);
```

- and a wrapper

```
char py_stableTimestep__name__[] = "stableTimestep";
PyObject * py_stableTimestep(PyObject *, PyObject * args)
{
    double dt = stableTimeStep("deformation");

    return Py_BuildValue("d", dt);
}
```

- one can place the result of the routine in a python variable

```
dt = danse.stableTimestep()
```

- The general case is not much more complicated than this

Support for distributed computing

- We are in the process of migrating the existing support for distributed processing into **`gsl`**, a new package that completely encapsulates the middleware

- Provide both user space and grid-enabled solution
- User space:
  - *ssh, scp*
  - *pyre service factories and component management*
- Web services
  - *pyGridWare from Keith Jackson's group*

- Advanced features
  - *dynamic discovery for optimized deployment*
  - *reservation system for computational resources*

Support for concurrent applications

- Python as the driver for concurrent applications that
  - *are embarrassingly parallel*
  - *have custom communication strategies*
    - *sockets, ICE, shared memory*
- Excellent support for MPI
  - **`mpipython.exe`**: *MPI enabled interpreter (needed only on some platforms)*
  - **`mpi`**: *package with python bindings for MPI*
    - *support for staging and launching*
    - *communicator and processor group manipulation*
    - *support for exchanging python objects among processors*
  - **`mpi.Application`**: *support for launching and staging MPI applications*
    - *descendant of* **`pyre.application.Application`**
    - *auto-detection of parallelism*
    - *fully configurable at runtime*
    - *used as a base class for user defined application classes*

Wrap up

- Contact info
  - *aivazis@caltech.edu*
- There is a lot of material on the web but it is disorganized
  - *currently at http://www.cacr.caltech.edu/projects/pyre*
  - *soon to be at http://pyre.caltech.edu*

Appendix: Writing a python extension module

- The Python documentation
- A library and headers
- A notion of how the Python scripts should look like
- The bindings
  - *the module entry point*
  - *the method table*
  - *the wrappers*
  - *error handling*

CSG – a simple solid modeler API

- Solid primitive constructors:

```
csg_body * csg_sphere(double radius);
csg_body * csg_cylinder(double radius, double height);
csg_body * csg_cone(double top, double bottom, double height);
csg_body * csg_block(double dx, double dy, double dz);
```

- Destructor:

```
void csg_destroy(csg_body *);
```

- Boolean operations:

```
csg_body * csg_unite(csg_body *, csg_body *);
csg_body * csg_subtract(csg_body *, csg_body *);
csg_body * csg_intersect(csg_body *, csg_body *);
```

- Transformations:

```
csg_body * csg_translate(csg_body *, double * displacement);
csg_body * csg_rotate(csg_body *, double angle, double * axis);
```

Python access to CSG

- We want to write scripts like:

```
import csg

sphere = csg.sphere(10)
cone = csg.cone(0, 7, 10)

cone = csg.translate(cone, (0,0,7))
jack = csg.unite(sphere, cone)
```

Anatomy of an extension module

- On **`import csg`**, the interpreter
  - *looks for a module named **`csg`** in the "standard places"*
  - *runs the module initializer that is responsible for*
    - *creating a module instance*
    - *populating the module method table*
    - *adding any other module-level attributes, if necessary*
- The method table establishes the association between Python names and function entry points
- When the name is used, the interpreter
  - *packs the arguments in a tuple*
  - *calls the binding*
  - *handles the return value*

The module file

```
// -- csgmodule.cc

#include <Python.h>

#include "solids.h"
#include "operations.h"
#include "transformations.h"
#include "exceptions.h"

static PyMethodDef csg_methods[] = {
    // see slide "The method table"
};

extern "C" void initcsg() {
    // see slide "The module entry point"
}

// End of file
```

The method table

```
static PyMethodDef csg_methods[] = {
// sanity
    {"hello", hello, METH_VARAGS, hello_doc},
// solids
    {"sphere", new_sphere, METH_VARARGS, sphere__doc},
    {"cylinder", new_cylinder, METH_VARARGS, cylinder__doc},
    {"cone", new_cone, METH_VARARGS, cone__doc},
    {"block", new_block, METH_VARARGS, block__doc},
// boolean operations
    {"unite", unite, METH_VARARGS, unite__doc},
    {"subtract", subtract, METH_VARARGS, subtract__doc},
    {"intersect", intersect, METH_VARARGS, intersect__doc},
//transformations
    {"rotate", rotate, METH_VARARGS, rotate__doc},
    {"translate", translate, METH_VARARGS, translate__doc},
// sentinel
    {0, 0}
};
```

The module entry point

- Minimal initialization

```
// The module initialization routine
extern "C" void initcsg()
{
    Py_InitModule(“csg", csg_methods);

    if (PyErr_Occurred()) {
        Py_FatalError("Can't initialize module csg");
        return;
    }

    return;
}
```

Sanity check

- Simple version

```
#include <iostream>

static PyObject * hello(PyObject *, PyObject *)
{
    std::cout << "Hello from csgmodule" << std::endl;

    Py_INCREF(Py_None); // the return value is None
    return Py_None;
};
```

- check

```
>>> from csg import hello
>>> hello()
Hello from csgmodule
>>>
```

Reference counts

- Python objects are not owned
- Instead, code elements have ownership of *references*
- Implemented using *reference counts*
- Manipulated using Py_INCREF and Py_DECREF
  - *no NULL checking, for speed*
  - *use the Py_XINCREF and Py_XDECREF variants when in doubt*
- The garbage collector currently relies on refcount
  - *when it reaches 0, the object's finalizer is called, if it exists*
  - *simple, fast, no "delay" effect*
  - *easy to defeat, e.g. circular references*
- Python 2.0 has a new garbage collector, but it is not yet the default

Why `Py_INCREF(Py_None)`?

- Consistency, consistency, consistency
- Simplified mental model:
  - *the return value of our function is stored in a temporary variable*
  - *the only way to access this value is to borrow references from the temporary variable*
  - *when the temporary is no longer usable, it will decrement the reference count*
    - *at the end of the statement*
    - *if an exception is thrown*
    - *…*
- We are creating an object to represent the return value of the function for our caller

Another sanity check

- Get an argument from the interpreter

```
#include <iostream>
static PyObject * hello(PyObject *, PyObject * args)
{
    char * person;
    if (!PyArg_ParseTuple(args, "s", &person)) {
        return 0;
    }

    std::cout << "csg: hello " << person << "!" << std::endl;

    Py_INCREF(Py_None); // the return value is None
    return Py_None;
}
```

- check

```
>>> from csg import hello
>>> hello("Michael")
csg: hello Michael!
```

The convenience functions

- PyArg_ParseTuple
  - *takes a format string and the addresses of variables*
  - *attempts to decode the args tuple according to the format*
  - *deposits the values in the variables*
  - *returns 0 on success, non-zero on failure*
- Py_BuildValue
  - *takes a format string and a value*
  - *builds the PyObject equivalent*
- Common codes: “s”, “i”, “l”, “d”
- Format codes and the syntax of the format string are described in the documentation

The bindings for the solid primitives

```
// -- solids.cc

#include <Python.h>

#include "solids.h"
#include "exceptions.h"

char sphere__doc[] = "Create a sphere of a given radius";

PyObject * new_sphere(PyObject *, PyObject * args)
{
    // see next slide
}

// The bindings for the other solid constructors

// End of file
```

The binding for `csg_sphere`

```
PyObject * new_sphere(PyObject *, PyObject * args)
{
    double radius;
    if (!PyArg_ParseTuple(args, "d", &radius)) {
        return 0;
    }
    if (radius < 0.0) {
        // Throw an exception
        return 0;
    }

    csg_body * body = csg_sphere(radius);
    if (!body) {
        // Throw an exception
        return 0;
    }

    return PyCObject_FromVoidPtr(body, csg_destroy);
}
```

Better error handling

- Improvements by:
  - *creating our own exception objects*
  - *install them in the module namespace*
  - *raise them when appropriate*
- Exceptions should be visible by all parts of the module
  - *they are intrinsically “global” objects for our module*
- In C they are global variables
- in C++ they can be attributes of a Singleton

Module exceptions

```
// The module initialization routine
extern "C" void initcsg()
{
    PyObject * m = Py_InitModule(“csg", csg_methods);
    PyObject * d = PyModule_GetDict(m);
    if (PyErr_Occurred()) {
        Py_FatalError("Can't initialize module csg");
        return;
    }

    TransformationException =
        PyErr_NewException(“csg.TransformationException",0,0);
    PyDict_SetItemString(d, te_doc, TransformationException);

    return;
}
```

Unpacking the arguments by hand

- You can get to the arguments directly
- **`args`** is a **`PyTuple`**

  `PyObject * PyTuple_GetItem(<tuple>, <position>)`
- For non-trivial argument lists, use the Python API to:
  - *count the number of arguments passed*
  - *iterate over the arguments*
  - *check that each argument is of the expected type*
- Use the Python conversion functions
    - *avoid casts (implicit or explicit)*
- Throw appropriate exceptions for bad arguments

The binding for `csg_translate` - I

```
PyObject * translate(PyObject *, PyObject * args)
{
    // Extract the body from position 0 in the argument tuple
    void * cobj = PyCObject_AsVoidPointer(PyTuple_GetItem(args, 0));
    csg_body *body = (csg_body *)cobj;
    if (!body) {
        // Throw an exception
        return 0;
    }

    PyObject *displacement = PyTuple_GetItem(args, 1);
    if (!PyTuple_Check(displacement)) {
        // Throw an exception
        return 0;
    }

// continued on the next slide
```

The binding for `csg_translate` - II

```
    // …
    // continued from the previous slide

    double v[3];
    v[0] = PyFloat_AsDouble(PyTuple_GetItem(displacement, 0));
    v[1] = PyFloat_AsDouble(PyTuple_GetItem(displacement, 1));
    v[2] = PyFloat_AsDouble(PyTuple_GetItem(displacement, 2));

    csg_body * result = csg_translate(body, v);
    if (!result) {
        // Throw an exception
        PyErr_SetString(TransformationException,"translate: ");
        return 0;
    }

    return PyCObject_FromVoidPtr(result, csg_destroy);
}
```

Creating a new type

- Not as well documented
  - *perhaps not common?*
- The main ingredients:
  - *The object record*
    - *a* `PyObject`*-compatible data structure to hold your object*
  - *The type record that describes the basic capabilities of the type*
    - *e.g., name, size, destructor, print function*
  - *A method table with the association between names and function entry points*
  - *A member table with the association between names and (types, offsets)*
  - *A few required overloads of the basic interface*
  - *A resting place for the constructor*
- Definitions and examples in the Python **source**

The object record

- Inherit from **`PyObject`**

```
// -- SolidBody.h

typedef struct {
    PyObject_HEAD
    csg_body * _brep;
} SolidBody;

// End of file
```

- **`PyObject`** and **`PyObject_HEAD`** live in *Include/object.h*

```
#define PyObject_HEAD \
         int ob_refcnt; \
         struct _typeobject *ob_type;

typedef struct {
    PyObject_HEAD
} PyObject;
```

The type record

- The type record also inherits from **`PyObject`**

```
PyTypeObject SolidBodyType = {
    PyObject_HEAD_INIT(&PyType_Type)
    0,
    "SolidBody",
    sizeof(SolidBody),
    0,

    destroy,      // destructor
    0, _          // print
    getAttr, 0,   // getattr, setattr
    0, 0,         // cmp, repr
    0, 0, 0,      // Object model protocols
    0, 0, 0,      // hash, call, str
    0, 0,         // getattro, setattro

    // others …
};
```

The constructor

```
PyObject * new_sphere(PyObject *, PyObject * args)
{   // see slide "The binding for csg_sphere"
    double radius;
    if (!PyArg_ParseTuple(args, "d", &radius)) {
        return 0;
    }

    csg_body * body = csg_sphere(radius);
    if (!body) {
        // Throw an exception
        return 0;
    }

    SolidBody * pybody = PyObject_New(SolidBody, &SolidBoyType);
    pybody->_brep = body;

    return (PyObject *)pybody;
}
```

The destructor

- Called when the reference count reaches zero
- Two tasks:
  - *destroy the csg object*
  - *deallocate the memory allocated by* ***`PyObject_New`***
- Casts galore …

```
void destroy(PyObject * arg)
{
    SolidBody * pybody = (SolidBody *) arg;
    csg_destroy(pybody->_brep);

    free(pybody);

    return;
}
```

The type method table

- From *Objects/fileobject.c*

```
static PyMethodDef file_methods[] = {
        {"readline",    (PyCFunction)file_readline, 1},
        {"read",        (PyCFunction)file_read, 1},
        {"write",       (PyCFunction)file_write, 0},
        {"fileno",      (PyCFunction)file_fileno, 0},
        {"seek",        (PyCFunction)file_seek, 1},
        {"tell",        (PyCFunction)file_tell, 0},
        {"readinto",    (PyCFunction)file_readinto, 0},
        {"readlines",   (PyCFunction)file_readlines, 1},
        {"writelines",  (PyCFunction)file_writelines, 0},
        {"flush",       (PyCFunction)file_flush, 0},
        {"close",       (PyCFunction)file_close, 0},
        {"isatty",      (PyCFunction)file_isatty, 0},
        {NULL,          NULL}            /* sentinel */
};
```

The type member table

- From *Objects/fileobject.c*

```
#define OFF(x) offsetof(PyFileObject, x)

static struct memberlist file_memberlist[] = {
         {"softspace",  T_INT,    OFF(f_softspace)},
         {"mode",       T_OBJECT,         OFF(f_mode),   RO},
         {"name",       T_OBJECT,         OFF(f_name),   RO},
         {"closed",     T_INT,    0,              RO},
         {NULL}  /* Sentinel */
};
```

Overloading `__getattr__`

- From *Objects/fileobject.c*

```
static PyObject *
file_getattr(PyFileObject *f, char *name)
{
    PyObject *res;

    res = Py_FindMethod(file_methods, (PyObject *)f, name);
    if (res != NULL) {
        return res;
    }
    PyErr_Clear();
    if (strcmp(name, "closed") == 0) {
        return PyInt_FromLong((long)(f->f_fp == 0));
    }

    return PyMember_Get((char *)f, file_memberlist, name);
}
```

Finishing touches: an OO veneer

- Why not create real Python classes
  - *Sphere, Cylinder, Cone,…*
  - *cache the constructor arguments*
  - *build the csg representation only when needed*
- What about the operations and transformations
  - *Patterns: Composite, Visitor, …*
  - *all in Python*
  - *cheap and fast*
- Is there added value?
  - *encapsulation of the csg engine*
  - *portability*

Writing extensions in C++

- A very active topic
  - *join the Python C++-SIG*
- Options:
  - *Automatic tools*
    - *not without a C++ parser on-board - ☹*
  - *Use Paul Dubois' CXX*
    - *open source*
    - *he is looking for someone to pick it up*
  - *Do it by hand (like I do)*
    - *Adapter/Bridge that inherits from PyObject and dispatches*
    - *suitable for few objects with stable public interfaces*

Embedding

- Not as well documented
  - *but the Python executable is an example!*
- The application entry point is under your control
- You have to initialize the interpreter
- Look at the “Very High Level” section of the manual for options
- Advantages:
  - *complete control over available modules*
  - *secure*
  - *Python version is frozen*
- Disadvantages
  - *You have to do everything yourself*
  - *Python version is frozen*

Resources

- The Python documentation
- The web at www.python.org
- The Python mailing lists
- The Python **source code**

Application strategies

- If you need a scripting language
  - *think of your users' spouses*
  - *please don't invent a new one*
- What can Python do for your application?
- How much Python can you afford NOT to have?
- How do you make performance an non-issue?
- …