Scientific Application Design

Michael Aivazis
California Institute of Technology
Distributed resources

Developers/users, Caltech

Visualization, Caltech

Development and post-processing, Caltech

Archival storage, Caltech

White, LLNL

Blue Mountain, LANL
Software engineering

- Large scale parallelism
  - Scalable algorithms
  - Scalable IO systems
  - But also:
    - debugging
    - monitoring
    - data management

- Team management
  - Source control with CVS
  - Automatic regular builds
  - Regression testing

- Evolving environments
  - Build procedure
    - Host, platform and compiler constraints
    - Developer choices
    - Package availability
Flexibility through the use of scripting

• Scripting enables us to
  – Organize the large number of application parameters
  – Allow the simulation environment to discover new capabilities without the need for recompilation or relinking

• The python interpreter
  – The interpreter
    • modern object oriented language
    • robust, portable, mature, well supported, well documented
    • easily extensible
    • rapid application development
  – Support for parallel programming
    • trivial embedding of the interpreter in an MPI compliant manner
    • a python interpreter on each compute node
    • MPI is fully integrated: bindings + OO layer
  – No measurable impact on either performance or scalability
Pyre: the integration framework

- An environment that enables the non-expert and does not hinder the expert

- Target features for “end-users”:
  - complete and intuitive simulation specification
  - reasonable defaults, consistency checks of input, good diagnostics
  - easy access to remote facilities
  - status monitoring

- Target features for “developers”:
  - easy access to user input
  - shorter development cycle, good debugging support

- Status as of 1/2002:
  - Python scripts: 1200 classes, 60,000 lines of code
  - C++: 25000 lines of bindings and infrastructure

- Recent success on nirvana:
  - 1764 processors, 2.7 mel solid fluid, 256x256x512 fluid
  - ~3k timesteps/24hrs
  - ~1.5 Tb of data
The scope of Pyre

- Problem specification
  - components and their properties
- Solid modeling
  - overall geometry
  - model construction
  - topological and geometrical information
- Boundary and initial conditions
  - high level specification
  - access to the underlying solver data structures in a uniform way
- Materials and constitutive models
  - materials properties database
  - strength models and EOS
  - association with a region of space
The scope of Pyre - continued

- **Solvers:**
  - selection and association with parts of the geometry
  - solver specific initializations
  - coupling mechanism specification
- **Simulation driver**
  - initialization
  - computation of appropriate time steps
  - orchestration of the data exchange
  - checkpoints and field dumps
- **Simulation instrumentation**
- **Integration of parallel/distributed programming infrastructure**
  - authentication, staging, data gathering and storage
- **Real-time visualization**
  - simulation monitoring for both local and remote jobs
- **Full simulation archiving**
Leveraging existing solvers

- Geometry
  - Meshing
  - Fem
  - Checkpoints
- Properties
  - Materials
  - Adv. features
  - Viz. support
- Python bindings
  - Mesher
  - Solver
  - StrengthModel
- Controller
- Application
- MPDb
- VTFApplication
def march(self, totalTime=0, steps=0):
    solver = self._solver

    # the main simulation time loop
    while 1:

        solver.startTimestep()

        solver.applyBoundaryConditions()
        solver.checkpoint()

        dt = solver.stableTimestep()
        solver.advance(dt)
        self._t = self._t + dt
        self._step = self._step + 1

        solver.endTimestep()

        # are we done?
        if steps and self._step >= steps: break
        if totalTime and self._t >= totalTime: break

        # notify solver that the simulation is finished
        solver.cleanup()

    return
Writing python bindings

• Given a “low level” routine, such as

```cpp
double adlib::StableTimeStep();
```

• and a wrapper

```cpp
char pyadlib_stableTimestep__name__[] = "stableTimestep";
PyObject * pyadlib_stableTimestep(PyObject *, PyObject * args)
{
    double dt = adlib::StableTimeStep("deformation");
    return Py_BuildValue("d", dt);
}
```

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

```python
dt = pyadlib.stableTimestep()
```

• The general case is not much more complicated than this
Integrated applications in Pyre

- Geometry
- pyacis
- ACIS
- MPDb
- MaterialModel
- Monitor
- Application
- Sensor/Probe
- Controller
- Solver
- InterfaceManager
- Adlib
- pyadlib
- solid engine
- ARM3d
- pyarm3d
- pygrace
- arm3d
- GrACE
- VTFApplication
- options
- script
- GUI
- pympi
- MPI
shock: a Pyre application
The main entry point

```python
# main program
options = pyre.applications.main(defaults, usage)

# detect invocation mode
if pyre.bool(options["worker"]):
    onComputeNodes(options)
else:
    onFrontEnd(options)
```

**main**
- reads the command line arguments
- compares them against **defaults**
- on errors, calls **usage()** to issue a help message
Command line arguments

> shock.py --help

Usage: ./shock.py [options ...]
Options: (default values in brackets)
  model:
    --model=<filename> [cube.py]
  parallelism:
    --nodes=<int> [2]
    --node-list=[<range>] []
    --ratio=<solid:fluid> [1:1]
  monitoring:
    --vizserver=<address> [asap.cacr.caltech.edu]
    --port=<int> [50000]
  advanced options:
    --solid=<solver name> [rigid]
    --fluid=<solver name> [pulse]
  reserved options for internal use:
    --worker=<bool> (manager/worker: false)
At the front end

Python embedded in an MPI application

```python
# launch the parallel job
def onFrontEnd(options):

    # construct the launching command
    # typical value on our cluster:
    # mpirun -np 100 `which mpipython.exe' shock.py --worker=true ...
    command = pyre.applications.launch(options)

    # execute it
    # could have used fork/exec instead
    import os
    os.system(command)

    # done
    return
```
Enabling parallelism in Python is implemented by:

- embedding the interpreter in an MPI application:

```c
int main(int argc, char **argv) {
    int status = MPI_Init(&argc, &argv);
    if (status != MPI_SUCCESS) {
        std::cerr << argv[0] << " : MPI_Init failed! Exiting ..." << std::endl;
        return status;
    }
    status = Py_Main(argc, argv);
    MPI_Finalize();
    return status;
}
```

- constructing an extension module with bindings for MPI
- providing an object oriented veneer for easy access
// return the communicator rank (MPI_Comm_rank)
char pympi_communicatorRank__doc__[] = "";
char pympi_communicatorRank__name__[] = "communicatorRank";
PyObject * pympi_communicatorRank(PyObject *, PyObject * args){

    PyObject * py_comm;
    int ok = PyArg_ParseTuple(args, "O:communicatorRank", &py_comm);
    
    if (!ok) {
        return 0;
    }

    // cast the Python object into a Communicator
    Communicator * comm = (Communicator *) PyCObject_AsVoidPtr(py_comm);
    int rank = comm->rank();

    // return
    return PyInt_FromLong(rank);
}
Access to MPI through Pyre

```python
import pyre

# get the world communicator
world = pyre.mpi.world()

# compute processor rank in MPI_COMM_WORLD
rank = world.rank()

# create a new communicator
new = 
if new:
    print "world: %d, new: %d" % (rank, new.rank())
else:
    print "world: %d (excluded from new)" % rank
```

creates a new communicator by manipulating the communicator group
At the compute nodes

```python
# launch the parallel job
def onComputeNodes(options):

    # place the command line options in an Options object
    options = setOptions(options)
    # create a Solver object
    solver = createSolver(options)
    # place this processor in the right communicator
    communicator = createCommunicator(options)
    # create a simulation controller
    controller = createController(solver, options)
    # initialize controller, solvers, etc.
    controller.launch(communicator)
    # run the simulation
    controller.march(steps=options.timesteps)

return
```
def createSolver(options):
    rank = pyre.mpi.world().rank()
    if rank in options.fluidProcessors:
        fluid = options.fluidSolver
        if fluid == "pulse":
            solver = pulse(options)
        elif fluid == "arm3d":
            solver = arm3d(options)
        else:
            raise "unknown fluid solver"
    elif rank in options.solidProcessors:
        solid = options.solidSolver
        if solid == "rigid":
            solver = rigid(options)
        elif solid == "adlib":
            solver = adlib(options)
        else:
            raise "unknown solid solver"
    else:
        raise "orphan processor"
    return solver
def pulse(options):
    pyre.debug.activate("pulse")
    pyre.debug.activate("pulse.timeloop")

    from pyre.units.length import mm
    from pyre.units.pressure import GPa
    from pyre.units.time import microsecond as us

    import pulse
    solver = pulse.solver(options)
    solver.verifyInterface()
    solver.defaultTimestep(0.5 * us)

    load = solver.generator("heaviside")
    load.amplitude = 10.0*GPa
    load.position = (0.0*mm, 0.0*mm, -51.0*mm)
    load.velocity = (0.0*mm/us, 0.0*mm/us, 1.0*mm/us)

    return solver
Instantiating a solid solver

```python
def rigid(options):
    pyre.debug.activate("rigid")
    pyre.debug.activate("rigid.timeloop")

    boundary, boundingBox = mesh(options.model)
    options.model = boundary

    import rigidObstacle
    solver = rigidObstacle.solver(options)
    solver.verifyInterface()
    solver.debugLevel(2)

    return solver
```
Creating the model

```python
def mesh(model):
    body = createBody(model)
    from pyre import acis
    faceter = acis.faceter()
    properties = faceter.properties()
    properties.gridAspectRatio = 1.0
    properties.maximumEdgeLength = body.ils
    boundary = faceter.facet(acis.create(body.geometry))
    bbox = boundary.boundingBox()
    return boundary, bbox
```

- **an abstract representation of the model**
- **the Python bindings for the solid modeler**
- **convert the abstract geometrical description into an actual instance of the ACIS class BODY**
def model():
    from pyre.units.length import mm
    side = 50.0*mm
    diameter = 25.0*mm
    scale = 5

    from pyre.geometry.primitives import block, cylinder
    from pyre.geometry.operations import rotate, subtract, translate

    cube = block((side, side, side))
    cube = translate(cube, (-side/2, -side/2, -side/2))
    hole = cylinder(height=2*side, radius=diameter/2)
    z_hole = translate(hole, (0*side, 0*side, -side))
    y_hole = rotate(z_hole, (1,0,0), pi/2)
    x_hole = rotate(z_hole, (0,1,0), pi/2)

    body = subtract(body, x_hole)
    body = subtract(body, y_hole)
    body = subtract(body, z_hole)

    ils = min(radius, side - diameter)/scale

    return pyre.geometry.body(body, ils)
Geometry specification – part II

- Difference
  - Rotation
    - Translation
    - Cylinder
  - Difference
    - Rotation
      - Translation
      - Block

- Body
The finished model
Introduction to Python

• Resources
• Interacting with the Python interpreter
  – *Interactive sessions*
• Overview of the Python language
  – *The focus of this presentation*
• Building Python extensions in C/C++
  – *Will be described in detail later*
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

• Local site:
  – www.cacr.caltech.edu/pyre
  – still under construction (might move)

• Mailing list: pyre-seminar@cacr.caltech.edu
  – still under construction
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**

<table>
<thead>
<tr>
<th>Type</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numbers</td>
<td>1234, 3.1415, 999L, 3+4j</td>
</tr>
<tr>
<td>Strings</td>
<td>‘help’, “hello”, “It’s mine”</td>
</tr>
<tr>
<td>Lists</td>
<td>['this', ['and', 0], 2]</td>
</tr>
<tr>
<td>Dictionaries</td>
<td>{'first': 'Jim', 'last': 'Brown'}</td>
</tr>
<tr>
<td>Tuples</td>
<td>(1, “this”, ‘other’)</td>
</tr>
<tr>
<td>Files</td>
<td>open('sample.txt', ‘r’)</td>
</tr>
</tbody>
</table>
Operators and precedence

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

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234, -1234</td>
<td>integers (C long)</td>
</tr>
<tr>
<td>9999L</td>
<td>arbitrary precision integers</td>
</tr>
<tr>
<td>3.1415, 6.023e-23</td>
<td>floats (C doubles)</td>
</tr>
<tr>
<td>0177, 0xdeadbeef</td>
<td>octal and hex constants</td>
</tr>
<tr>
<td>j, 1.0 – 3.14j</td>
<td>complex numbers</td>
</tr>
</tbody>
</table>
Strings

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>s = &quot;&quot;</td>
<td>empty string</td>
</tr>
<tr>
<td>s = “It’s mine”</td>
<td>double quotes</td>
</tr>
<tr>
<td>s = &quot;&quot;&quot;&quot; ... &quot;&quot;&quot;&quot;</td>
<td>triple quote blocks</td>
</tr>
<tr>
<td>s1 + s2, s1 * 4</td>
<td>concatenate, repeat</td>
</tr>
<tr>
<td>s[i], s[i:j], Len(s)</td>
<td>index, slice, length</td>
</tr>
<tr>
<td>for x in s, ‘m’ in s</td>
<td>iteration, membership</td>
</tr>
</tbody>
</table>
Lists

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

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

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>d = {}</td>
<td>empty dictionary</td>
</tr>
<tr>
<td>d1 = {'last': 'Brown', 'height': 1.85}</td>
<td>Two items</td>
</tr>
<tr>
<td>d2 = {'person': {'last': 'Brown', 'height': 1.85}, 'state': 'dead'}</td>
<td>nested dictionaries</td>
</tr>
<tr>
<td>d1['last'], d2['person']['last']</td>
<td>indexing</td>
</tr>
<tr>
<td>d.has_key('last')</td>
<td>method</td>
</tr>
<tr>
<td>len(d)</td>
<td>number of entries</td>
</tr>
<tr>
<td>del d[key], d[key] = value</td>
<td>remove, add/change</td>
</tr>
</tbody>
</table>
Tuples

- Immutable ordered sequences of object references
- Common operations:

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

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

<table>
<thead>
<tr>
<th><strong>Operation</strong></th>
<th><strong>Meaning</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>out = open(&quot;hello.txt&quot;, &quot;w&quot;)</code></td>
<td>create output file</td>
</tr>
<tr>
<td><code>in = open(&quot;hello.txt&quot;, &quot;r&quot;)</code></td>
<td>create input file</td>
</tr>
<tr>
<td><code>in.read()</code>, <code>in.read(1)</code></td>
<td>read file, read byte</td>
</tr>
<tr>
<td><code>in.readline()</code>, <code>in.readlines()</code></td>
<td>read a line, fill a list of strings</td>
</tr>
<tr>
<td><code>out.write(s)</code>, <code>out.writelines(L)</code></td>
<td>write a string, write a list of strings</td>
</tr>
<tr>
<td><code>out.close()</code></td>
<td>flush and close file explicitly</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Object Type</th>
<th>Category</th>
<th>Mutable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numbers</td>
<td>Numeric</td>
<td>No</td>
</tr>
<tr>
<td>Strings</td>
<td>Sequence</td>
<td>No</td>
</tr>
<tr>
<td>Lists</td>
<td>Sequence</td>
<td>Yes</td>
</tr>
<tr>
<td>Dictionaries</td>
<td>Mapping</td>
<td>Yes</td>
</tr>
<tr>
<td>Tuples</td>
<td>Sequence</td>
<td>No</td>
</tr>
<tr>
<td>Files</td>
<td>Extension</td>
<td>Yes</td>
</tr>
</tbody>
</table>
String coercions and the format operator

- All objects can be represented as strings
  - using the `__str__` operator
  - using the `repr()` built-in function
  - using the `str()` built-in function

- The string format operator ‘%’
  - the Python equivalent of `printf`
  - 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:

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

Numbers
- Integers
  - String
  - Tuple
- Float
- Complex

Containers
- Sequences
  - Immutable
    - String
    - Tuple
  - Mutable
    - List
    - Tuple
- Mappings
  - Dictionary

Other
- Module
- Instance
- File
- None

Internals
- Type
- Code
- Frame
- Traceback

Callable
- Function
- Method
- Class

Bound
Unbound
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 – but Guido wants to take this away …
## Reserved words

<table>
<thead>
<tr>
<th>access</th>
<th>and</th>
<th>break</th>
<th>class</th>
</tr>
</thead>
<tbody>
<tr>
<td>continue</td>
<td>def</td>
<td>del</td>
<td>elif</td>
</tr>
<tr>
<td>else</td>
<td>except</td>
<td>exec</td>
<td>finally</td>
</tr>
<tr>
<td>for</td>
<td>from</td>
<td>global</td>
<td>if</td>
</tr>
<tr>
<td>import</td>
<td>in</td>
<td>is</td>
<td>lambda</td>
</tr>
<tr>
<td>not</td>
<td>or</td>
<td>pass</td>
<td>print</td>
</tr>
<tr>
<td>raise</td>
<td>return</td>
<td>try</td>
<td>while</td>
</tr>
</tbody>
</table>
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

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

• The obligatory “Hello world” program:

```python
print “Hello world”
```
Assignments

• Explicitly, using =

\[
\text{<name>} = \text{<expression>}
\]

• Implicitly, using import, def, class

\[
\text{import \ <module>}
\]
\[
\text{from \ <module> import \ <name>}
\]
\[
\text{from \ <module> import \ <name> \ as \ <alias>}
\]
\[
\text{def \ <name>(<parameter \ list>):}
\]
\[
\text{class \ <name>(<ancestor \ list>):}
\]
Selections

• Using if

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

• No switch statement
  • use if
  • or lists and dictionaries
Explicit loops

- **while**

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

- **for**

  ```python
  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:**

  ```python
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:**

  ```python
def isServer(processor_id):
  if processor_id is 0: return True
  return False
  ```
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

```python
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.

```python
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`

```python
import string

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

```python
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

```python
from pyre.support.debug import DebugCenter
```
Namespaces as objects

• Modules and packages are objects:

```python
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

```python
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*

```python
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:
  - Calling methods through instances: `object.method(arguments...)`
  - Calling methods through classes: `Class.method(object, arguments...)`
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:

<table>
<thead>
<tr>
<th>Method</th>
<th>Overloads</th>
<th>Method</th>
<th>Overloads</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td>Construction: ( x = X() )</td>
<td><strong>getattr</strong></td>
<td>Qualification: ( x.\text{undefined} )</td>
</tr>
<tr>
<td><strong>del</strong></td>
<td>Destruction</td>
<td><strong>getitem</strong></td>
<td>Indexing: ( x[5] )</td>
</tr>
<tr>
<td><strong>repr</strong></td>
<td>Representation: <code>\( x \)</code>, ( \text{repr}(x) )</td>
<td><strong>setitem</strong></td>
<td>Indexing: ( x[5] = 0 )</td>
</tr>
<tr>
<td><strong>str</strong></td>
<td>String coercion: ( \text{str}(x) )</td>
<td><strong>add</strong></td>
<td>Addition: ( x + \text{other} )</td>
</tr>
<tr>
<td><strong>len</strong></td>
<td>Size, truth tests: ( \text{len}(x) )</td>
<td><strong>radd</strong></td>
<td>Addition: ( \text{other} + x )</td>
</tr>
<tr>
<td><strong>cmp</strong></td>
<td>Comparisons: ( x &lt; \text{object} )</td>
<td><strong>and</strong></td>
<td>Logic: ( x ) and ( \text{object} )</td>
</tr>
<tr>
<td><strong>call</strong></td>
<td>Function calls: ( x() )</td>
<td><strong>or</strong></td>
<td>Logic: ( x ) or ( \text{object} )</td>
</tr>
</tbody>
</table>
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!*

©2001-2002, Michael Aivazis
Classes as objects

- Classes are objects
- Examples:

```python
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

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

method(object, arguments)
```

• Bound method objects

```python
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:

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

```python
try:
    <statements>
finally:
    <statements>
```
Writing an 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:
  
  ```c
  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:

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

• Boolean operations:

  ```c
  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:

  ```c
  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:

```python
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
static PyMethodDef csg_methods[] = {
    // sanity
    {"hello", hello, METH_VARARGS, 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

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

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

    return;
}
```


Sanity check

- Simple version

```cpp
#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

```python
>>> 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

```cpp
#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

```python
>>> 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`

```c
PyObject * new_sphere(PyObject * body, 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
/ 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 PyTuples**
  ```c
  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`

```c
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 \texttt{PyObject}

```c
// -- SolidBody.h
typedef struct {
    PyObject\_HEAD
    csg\_body * _brep;
} SolidBody;
```

- \texttt{PyObject} and \texttt{PyObject\_HEAD} live in \texttt{Include/object.h}

```c
#define PyObject\_HEAD
    int ob\_refcnt; \
    struct \_typeobject *ob\_type;
typedef struct {
    PyObject\_HEAD
} PyObject;
```
The type record

- The type record also inherits from \texttt{PyObject}

```c
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,          // getattr, setattr

    // others ...
};
```
The constructor

```c
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*

```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*

```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

```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?
• …
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

- **Base**
  - **class name**
  - **attributes**
  - **operations**

- **Class**
  - **attributes**
  - **operations**

**Access**
- + public
- = protected
- - private

**Attributes**
- access **name** : **type** = **initial value**

**Operations**
- **method** *(argument list)* : **return type**

**name** : **type** = **initial value**
Solid primitives

- Here is a possible class hierarchy

- **Body**
  - `__init__()`

- **Block**
  - `__init__()`
  - `_diagonal`

- **Sphere**
  - `__init__()`
  - `_radius`

- **Cone**
  - `__init__()`
  - `_top`
  - `_bottom`
  - `_height`

- **Cylinder**
  - `__init__()`
  - `_radius`
  - `_height`

- **Torus**
  - `__init__()`
  - `_major`
  - `_minor`

- **Pyramid**

- **Prism**
# Operations

- What is the best way to represent these?

<table>
<thead>
<tr>
<th>Intersection</th>
<th>Union</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>- _op1</td>
<td>- _op1</td>
<td>- _op1</td>
</tr>
<tr>
<td>- _op2</td>
<td>- _op2</td>
<td>- _op2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rotation</th>
<th>Translation</th>
<th>Reflection</th>
<th>Dilation</th>
<th>Reversal</th>
</tr>
</thead>
<tbody>
<tr>
<td>- _angle</td>
<td>- _vector</td>
<td>- _vector</td>
<td>- _scale</td>
<td></td>
</tr>
<tr>
<td>- _vector</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
“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:

```
    Body
     |
     |
    Composite
     |
     |
    Binary
```

```
    Primitive
     |
     |
    Transformed
```

implementation?
The class hierarchy

- **Body**
  - **Composite**
    - **Transformed**
      - Rotate
      - Translate
      - Reflect
      - Dilate
    - Reverse
  - **Binary**
    - Union
    - Difference
    - Intersection
  - **Primitive**
    - **Block**
    - **Sphere**
    - **Cylinder**
    - **Cone**
    - **Pyramid**
    - **Prism**
    - **Torus**
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 define a new operation without changing the classes of the elements on which it operates”

```
Component
  Composite
  Leaf

Visitor
  OpA
  OpB
```