Python as a Discrete Event Simulation environment

F. Oliver Gathmann

University of Toronto, Surface and Groundwater Ecology Research Group,
Division of Life Sciences, 1265 Military Trail, M1C 1A4 Scarborough, Ontario, Canada
gathmann@scar.utoronto.ca

Abstract

This paper summarizes experiences made with using Python for implementing an interactive environment for Discrete Event Simulation of patchy animal populations. The main novel features of the system are runtime assembly of the simulation object sources allowing for easy integration of user-defined code, flexible monitoring of the simulation process using fast numerical arrays of variable dimensionality (based on the Numerical Python extension), and realtime multivariate analysis of the simulation results.

1 Background

The original aim of the project presented in this paper was to provide a specialized and yet customizable environment for the discrete event simulation (DES) of patchy animal populations ('metapopulations'). Being primarily concerned about a short development time rather than maximum runtime performance, and constrained by a limited budget that precluded the acquisition of a professional simulation package, I looked for a simple yet powerful, feature-rich yet affordable scripting language to get the project under way – and found that Python met all these requirements exceptionally well. As the system evolved, more and more flexibility was added and the current version of the '*inter-site*' simulator, although still predominantly aiming at the simulation of ecological populations and communities in a spatially realistic setting, features many elements of a general-purpose simulation environment.

Traditionally, DES has been the domain of compiled languages, mostly because of performance reasons. The simulation model and corresponding parameter settings had to be specified in advance of the actual (batch mode) simulation run and the results could be analyzed only after the latter had finished.

Clearly, interactivity during the simulation process would greatly enhance the productivity of any simulation environment compared to a traditional batch mode system as the user could

- stop, save and re-run parts of the simulation;
- efficiently fine-tune parameter settings;
- modify the structure of the simulation by changing, adding or removing parts;
- perform real-time statistical analyses on the results.

Only a relatively recent breed of simulation environments provides full interactivity in the above sense. This is typically achieved by providing runtime access to an abstract class library of generic simulation objects in one (or several) of the following ways (for an overview of professional systems see [Mau98]):

1. via a sophisticated graphical user interface (GUI; e.g., [RV93, LH94]);
2. via an existing interactive system for scientific computing (e.g., MATLAB/SIMULINK [The98]);
3. via a customized interpreted scripting language (e.g., SIMPLE++ [Tec98]).

The latter solution is related to a technique called 'steering' that is rapidly gaining popularity in scientific computing (overview: [YFD97], examples: [Hin97, BL97]). In steering, a general-purpose scripting language is used to provide a flexible interface to a collection of compiled assets, thus combining the best of both worlds.

My approach to developing an interactive simulation package combines elements from several of the techniques listed above in that I started off with a general-purpose scripting language, heavily exploiting the benefits offered by an interpreted environment, and only later improved the performance of the system by moving numerically expensive routines into a C extension of the Python interpreter (cf. [DY96, vR97]).

The general architecture of the *inter-site* system and experiences with using it as a tool in population simulation are explored elsewhere [GWss]; instead, this paper focuses on some of the more Python-specific issues involved in the implementation and use of the simulator.

2 Elements of DES

The *inter-site* simulator adopts the event-scheduling world view of DES, a well-known technique which is based on three generic runtime components (see Fig. 1; cf. [Eva88, Fis95] for detailed accounts on DES):

1. entity set: a collection of simulation entities that reflect the static structure of the simulated real-world system;

2. event queue: a list of events that realize the dynamics of the simulated real-world system as a series of state changes in the corresponding simulation entities, ordered by their scheduled execution time (time-stamp);

3. monitors: a set of probes that collect data about the simulation by translating individual simulation events into changes of state variables.

The actual simulation proceeds by de-queueing an event from the event queue, notifying the monitors about it, and performing discrete state changes in its associated entity (the 'action' of the event). Events can trigger one or several new events, possibly depending on the state of selected entities (the 'condition' for an event).

DES has been very popular since the early '70s and successful applications abound in a large number of very different fields in business (e.g., traffic control, consumer behavior forecasting, manufacturing, pest control: [Pri95]) and science (e.g., biochemistry: [RML94], population biology: [HCV97]).

3 Mapping real-world objects into Python objects for DES

The key to successful DES of any real-world system is the initial mapping of its static and dynamic structure into corresponding sets of simulation entities and events, respectively. Object-oriented programming (OOP) languages are a natural choice for this procedure since they are specifically designed to reflect the structure and behavior of real-world objects.

Structural descriptions of real-world objects can effectively be expressed in terms of their *attributes*. At any given point in time, each attribute of an object has a certain value, and the total set of attribute values of an object defines its current *state*. An attribute has either *specific* or *generic* scope (i.e., it applies to one particular object or to a set of objects belonging to the same class), and it is either *static* or *dynamic* (i.e., its value stays constant or varies in time).

Behavioral descriptions of objects can be obtained from observing changes in the values of dynamic at-

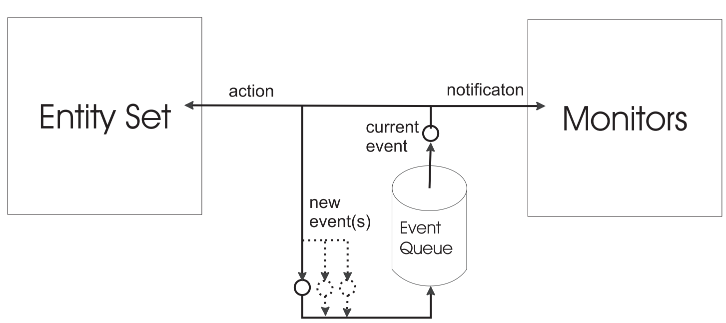

Figure 1: Generic components of a DES system

tributes (i.e., state changes) over time. In modeling dynamic attribute changes, an additional technical distinction is useful between attributes that take on a new value in a direct transition and attributes that return a new value as an indirect reflection of changes in other dynamic attributes (*transitive* and *reflexive* dynamic attributes, respectively).

Python's flavor of OOP provides a particularly neat and seamless way of mapping these different categories of real-world object descriptions directly into source code which is demonstrated in Fig. 2 using an object that should be familiar to every Python user:

All real members of the species *Python regius* lack legs and they feature a (more or less) banded skin patterning; these are generic attributes which are mapped into the class scope of the corresponding Python object. The time of birth as well as body temperature and body size are specific attributes of each *P. regius* individual and are therefore mapped into the instance scope of the Python object. The (constant) value of static instance attributes – like time of birth – is set during initialization in the **`__init__`** function. Transitive dynamic attributes are conveniently handled in the **`__call__`** function; in the example, we set the body temperature to the outside temperature as this is typical for poikilotherme animals like pythons, and we even let the class attribute **`pattern_type`** be changed in a spontaneous mutation when the body temperature exceeds a certain value, resulting in an improved ability of all future **`python_regius`** instances to dissipate heat because of a solid black skin patterning (an unlikely case of Lamarckian evolution). Finally, reflective attributes are handled in the **`__getattr__`** function, as it is shown for **`body_size`**, which we assume to increase linearly with age.

This straightforward mapping scheme was extensively used in solving the problem of how to provide runtime flexibility of simulation objects.

4 Simulation object code assembly

To maximize the interactivity of the system, it had to be possible not only to modify existing simulation entity objects on the fly by adding new attributes, but also to allow for runtime-creation of corresponding new simulation events affecting them.

As stated above, in current simulation environments this problem is typically solved by providing a library of generic simulation objects which are used to assemble custom objects via inheritance and/or composition at runtime. Instead of developing yet another general purpose simulation library, and intrigued by the possibility of importing arbitrary Python code into the running interpreter, I decided to take on a slightly different approach using what could be called *assembly from composite templates.*

In OOP, the term 'template' applies to classes that receive a type parameter during initialization (e.g. [KL89]). In the *inter-site* simulator, templates are composite structures that consist of a *prototype*, one or several *modules* and a *profile*. Prototypes implement the builtin-functionality of simulation objects. Modules are user-defined code segments adding to this basic functionality – new attribute definitions

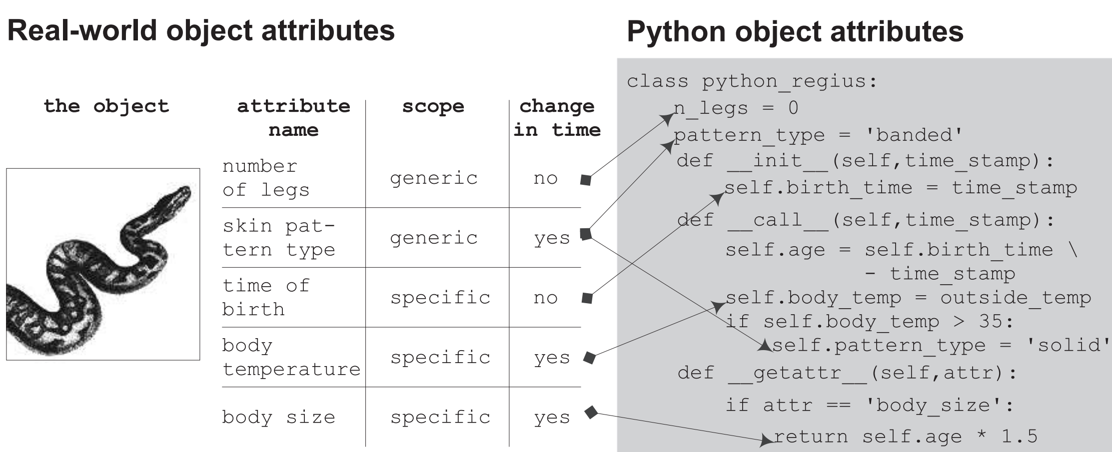

Figure 2: Mapping real-world objects into Python objects

for entity prototypes and new action code for event prototypes, respectively. Finally, profiles are used to provide initialization values for the attributes of a particular prototype-module(s) combination (parameterization).

Using this scheme, runtime definition of simulation objects is a matter of composing templates, i.e. selecting a prototype and adding (or selecting from existing) module and profile definitions according to the desired object specifcations. The Python code implementing the simulation objects is then assembled automatically from these template definitions and loaded into the running interpreter.

Before the actual source code assembly of a user-composed template can take place, its runtime relationships to other templates used for a particular simulation run have to be specified in a structure called *template set*. A template set contains

- for entity templates: information about runtime connectance with other entity templates (e.g., hierarchical as in `Individual` and `Population`) and specification of the first event to affect an entity template (if any), to be scheduled upon initialization of the latter;
- for event templates: event sequence definitions for dynamical triggering of new events.

A simple example should clarify the procedure. If `<templateD>` was a dictionary holding runtime template declarations, simple entries for an entity and an event template would look like this, respectively:

```
# entity:
templateD['Python'] = \
  { 'name' : 'Python' ,
    'prototype' : Individual_Proto ,
    'moduleL' : [],
    'profile' : 'standard'
    }
# event:
templateD['Move'] = \
  { 'name' : 'Move' ,
    'prototype' : Move_Proto ,
    'moduleL' : [] ,
    'profile' : 'standard'
    }
```

With these definitions, the simulation entity `Python` would be assembled from the `Individual_Proto` prototype (a class implementing entities of the 'Individual' type) and it would use the `'standard'` profile for parameterization of its attributes. The same scheme is used for assembling the simulation event template `Move`, which implements simple straight-line movements for the corresponding entity. Neither of the two templates uses any modules.

A simplified version of the corresponding prototypes will look like this:

```
# entity:
class Individual_Proto(Generic_Entity):
    class_attrD = \
      { 'mv_length' : '1' ,
        'p_survival' : '0.99' }
    instance_attrD =
      { 'mv_angle' : 'rad(rndint(0,359))' ,
        'pos' : '(0,0)' }
# event:
class Move_Proto(Generic_Event):
    action_codeD = \
      { 'newpos' : 'pos =
         pos[0]+cos(mv_angle)*mv_length,
         pos[1]+sin(mv_angle)*mv_length)' }
    condition_functionD = \
      { 'die' : 'p_survival < rnd()' }
```

Both entity and event prototypes inherit from generic classes which provide the internal functionality needed for setting up relationships among entities and for scheduling events.

Note that the values in the attribute dictionaries above are providing defaults for the resulting template, which can be overridden by module and/or profile specifications.

The key `'newpos'` in the action code dictionary serves as an identifier for the corresponding action code string (an action *atom*). Using these identifiers, atoms provided by the prototype can be re-defined in modules (mimicking polymorphic inheritance), and the execution sequence for a set of atoms can be specified by defining interdependencies among them.

Profiles for the two templates above would simply consist of a dictionary pairing each attribute name string with a new attribute value string; in the example, the defaults provided by the prototype will be used for simplicity.

The two template definitions shown above would result in the following (again, slightly simplified) source code skeleton:

```
# entity:
class Python(Generic_Entity):
    mv_length = 1
    p_survival = 0.99
    def __init__(self,<args>):
        <internal initialization code>
        self.mv_angle = rad(rndint(0,359))
        self.pos = (0,0)
# event:
class Move(Generic_Event):
    def __init__(self,<args>):
        <internal initialization code>
    def __call__(self):
        self.ind.pos = \
            (self.ind.pos[0] +
             cos(self.ind.mv_angle) *
             self.ind.mv_length,
             self.ind.pos[1] +
             sin(self.ind.mv_angle) *
             self.ind.mv_length)' }
    def die(self):
        return self.ind.p_survival < rnd()
```

In contrast to the code from the more generalized mapping scheme presented in Fig. 2, in the template code the `__call__` function used for transitions of dynamic entity attributes has moved into an independent event object. This is a direct consequence of the DES paradigm stating that entity state changes are mediated by events and is crucial for the generation of rich runtime behavior of simulation entities.

The pointer `ind` used in the `__call__` function of the event template is set up during initialization. The functions `rndint`, `rad`, `sin` and `cos` are public functions for obtaining a random integer, conversion of degree to radian, sinus, and cosinus, respectively.

The initialization code provided from the generic base class of the prototype is inserted into the `__init__` method of the template to avoid multiple constructor calls. In typical DES applications involving instantiation of many thousands of entity and event objects, this will amount to notable time savings.

Finally, the following is a template set specifying that a single python individual be sent off on a lonely straight-line trajectory for, on average, 100 moves (since `p_survival` is set to 0.99):

```
template_setD = \
  { Python :
     { 'init_event' : Move } ,
    Move :
     { 'event_sequence' :
         { Move.die : None ,
           'default' : 'continue' }
     }
    }
```

Upon instantiating the single `Python` instance, its `init_event` (`Move`) is initialized and scheduled for execution. Generation of subsequent events is governed by the `event_sequence` attribute of the `Move` event. This most simple example of an event sequence reads as follows: 'If the condition function `Move_1.die` returns false, reschedule the current event (with an incremented time stamp), if it returns true, return `None` (here, equivalent to the end of the simulation)'. Event sequences can contain arbitrary numbers of branches as well as logical conjunctions of several condition functions in a single branch for more complex conditions. Typically, a new event template instance is returned after evaluation of an event sequence instead of the 'special' return values `None` and `'continue'` shown above.

Compared to the traditional inheritance/composition mechanism, the source code assembly technique introduced above offers

- easy integration of user-defined code: by specifying the semantic context of the code additions, the user can experiment with the actual source code representation of the simulation objects;
- modularity within *one* method call: using several modules for one template still results in one method call (`__init__` for entity initialization and `__call__` for event actions, respectively).

Of course, an open template system like this can only work if the user-introduced code conforms to the general framework used for the implementation of the templates. For example, referring to a non-existing entity attribute in event action code would raise an `AttributeError` during the simulation. However, since the total set of attributes of each entity template is known in advance from the prototype and the module declarations, preventive checking for attribute errors is possible. This has been implemented using the `tokenize` module of the standard Python library, which offers a convenient way of parsing Python code into tokens and process the latter with a customized `tokeneater` function.

Augmenting existing prototypes in the way described above can be done with only minimal programming experience in Python. Composing new prototypes requires a little more insight in the internals of the *inter-site* system, but is fairly standardized in that

- the internal functionality is automatically inherited from the generic entity and event classes;
- traditional inheritance from existing prototypes facilitates the derivation of more specialized prototypes (e.g., for refining a 'Python' prototype into a 'Burmese Python' prototype).

Even more ambitious users might want to increase the overall performance of the simulation by replacing 'hardened' (Python) module code with C extension functions. This is a relatively straightforward task for experienced Python programmers given the clear specifications of the Python-C API and the restricted way of attribute access within the *inter-site* framework.

As shown in the examples above, the template, module and profile definitions consist mainly of code and documentation strings kept in simple dictionaries, so that storing them to disk for later reuse is easily achieved using the `pickle` and `shelve` services from the Python standard library.

5 Monitoring

Monitoring, the third major component of DES (cf. Fig. 1), faces special challenges in a highly flexible simulation environment like the one presented here. To list but a few of them, carrying on the ecological example introduced above:

- the number of state variable values to be recorded is often unknown (e.g., times of individual movement events);
- the number of objects to be monitored might change as the simulation proceeds (e.g., individuals die and reproduce);
- attributes added interactively to entities might affect the dimension of some monitors (e.g., we might add a `generation` attribute to the `Python` template and then want to monitor different generations separately).

With regard to the data structure used for monitoring, these three examples boil down to the problem of providing fast access to homogeneous numerical arrays of variable dimension. Fortunately, Python offers a very powerful extension module for manipulation of homogeneous numerical arrays at nearly the speed of compiled code (Numerical Python or `NumPy`, [HHD98]). By writing a wrapper class around a numerical array with customized `__getitem__` and `__setitem__` methods, I built a generic monitor data array that offers two different mechanisms for dynamical resizing:

- by *extending*: in this mode, data are stored by indexing or slicing operations and the array is dynamically resized whenever an `IndexError` occurs;
- by *appending*: this mode simulates an n-dimensional array of lists ('ragged array') by maintaining an $(n-1)$-dimensional array of counters of actual 'list' lengths. Data are stored by calls to an `append` method, which determines the actual slice corresponding to the append request using the counter array. As above, an `IndexError` in the subsequent indexing or slicing operation triggers dynamical resizing of the whole array.

Each monitor data array receives an axis descriptor dictionary upon intialization which provides a name, the initial size, and the resizing mode ('disabled' [i.e., fixed length], 'extend', or 'append') for each axis, thus allowing for some optimizations (e.g., avoiding overhead for resizing mechanisms that are not used by any of the axes) and for accessing the monitor array data by axis name (by substituting axis name strings with the corresponding array dimensions in the slice passed to `__getattr__`).

The *inter-site* system provides routines for flexible runtime definition of monitors, thereby mirroring the plasticity of the monitored simulation objects. The key requirement here is to ensure that any simulation event can be translated into an arbitrary change of the value of the monitored state variable depending on both the state of the corresponding simulation entity and the current value of the same or some other state variable (for example, a 'Death' event should increment the monitored state variable 'number of population extinctions' only if the population size reaches zero with this very event). This problem was solved by developing a 'monitor API' of functions for querying, slicing and updating monitor arrays that are used together with `getattr` calls querying the entity object state to specify monitoring activities at runtime in the way described above. Like template definitions for the simulation objects, the monitor definitions consist mainly of code strings that can easily be stored to disk for later reuse.

6 Realtime multivariate analysis of results

Here again the Numerical Python module shines: two of the more common dimension-reduction techniques for multidimensonal data, Principal Components Analysis (PCA) and Correspondence Analysis (CA) require little more than an eigen-decomposition of the input matrix (a sample-by-variable table), which is readily provided by the `lapackmodule` that comes with the `NumPy` distribution. The *inter-site* simulator offers an animated view on the optimal two-dimensional subspace of the simulation data derived by PCA or CA analysis in user-defined time intervals ('animated ordinations'). For data plotting, the highly versatile Graph-widget of the blt 2.4 library [How98] is used via the interface provided by Pmw-blt [McF97].

7 Experiences and availability

In a first medium-scale test application, the *inter-site* system has been used to re-implement a state-variable model on metapopulation persistence from the literature as an individual-based model (see [GWss] for details). A typical simulation run followed some 2000 individuals, each of which completed a full life cycle of development, dispersal, reproduction, and death per generation in an artificial landscape, over 300 generations and took roughly half an hour to complete on a 400Mhz DEC Alpha station. I have not been able to compare these numbers to the performance of any professional package for a similar simulation model. However, the number of individuals and generations employed was perfectly satisfactory for this medium-scale ecological application, and so was the reported average CPU time used for a single run. Furthermore, given that most of the time during simulating a real-world system is spent on slight to moderate modifications of the attributes and behavior of the simulation objects in order to gain deeper insight into the system's structure and dynamics, I am confident that I saved as much time using Python and the convenient code-assembly facilities of the *inter-site* system during repeated development cycles as I lost during the actual simulation runs.

Unfortunately, the first public release of the *inter-site* simulator will be not available until early 1999, as various other obligations distract me from completing the documentation and the GUI.

8 Conclusion

In this project, the Python language has proven to be an excellent choice for the implementation of a flexible DES package in a research environment, where typical users will have at least basic knowlegde in programming and are likely to be interested in experimenting with their own extensions to whatever pre-defined simulation objects are available.

Clearly, the *inter-site* system in its current version does neither provide the convenience nor the performance of commercially available large-scale simulation environments. However, it is certainly possible to transfer some of the techniques presented in this paper, namely code assembly from composite templates in combination with flexible monitoring and real-time data analysis services, to a more performance-oriented system based on a generic C++ library (e.g., [KL94]).

Acknowledgments

This study was supported by a HSP-II-AUFE scholarship of the DAAD, Germany, by the Schmeil Stiftung, Germany, and by the Natural Sciences and Engineering Research Council of Canada.

References

[BL97] D. M. Beazley and P. S. Lomdahl. Feeding a large-scale physics application to Python, 1997. Proc. Sixth Int. Python Conf. 1997. URL http://www.python.org/workshops/1997-10/proceedings/beazley.ps.

[DY96] P. F. Dubois and T.-Y. Yang. Extending Python. *Computers in Physics*, 10(4), 1996.

[Eva88] J. B. Evans. *Structures of discrete event simulation.* Ellis Horwood Limited, Chichester, 1988.

[Fis95] P. A. Fishwick. *Simulation model design and execution: building digital worlds.* Prentice Hall, Englewood Cliffs, NJ, 1995.

[GWss] F. O. Gathmann and D. D. Williams. *Intersite*: A new tool for the simulation of spatially realistic population dynamics. *Ecological Modelling*, in press.

[HCV97] D. Hill, P. Coquillard, and J. Vaugelas. Discrete-event simulation of alga expansion. *Simulation*, 68(5):69–277, 1997.

[HHD98] J. Hugunin, K. Hinsen, and P. Dubois. Numerical Python, 1998. URL ftp://ftp-icf.llnl.gov/pub/python/.

[Hin97] K. Hinsen. The molecular modeling toolkit: a case study of a large scientific application in Python, 1997. Proc. Sixth Int. Python Conf. 1997. URL http://www.python.org/workshops/1997-10/proceedings/hinsen.ps.

[How98] G. Howlett. blt library, release 2.4f, 1998. URL ftp://ftp.tcltk.com/pub/blt/.

[KL89] W. Kim and F. H. Lochovsky, editors. *Object-oriented concepts, databases and applications.* ACM Press, New York, 1989.

[KL94] H. Kocher and M. Lang. An object-oriented library for simulation of complex hierarchical systems. In *Proceedings of the Object Oriented Simulation Conference 1994 (OOS '94)*, pages 145–152. IEEE Computer Society, 1994.

[LH94] L. Y. Liu and C. H. Herring. Integration of interactive graphics and discrete-event simulation using object-oriented programming. In *Proceedings of the Object Oriented Simulation Conference 1994 (OOS '94)*, pages 171–175. IEEE Computer Society, 1994.

[Mau98] R. Mauth. Better simulation software, 1998. URL http://www.byte.com/art/9803/sec16/art1.htm.

[McF97] G. McFarlane. Python mega widgets for tk, 1997. URL http://www.dscpl.com.au/pmw.

[Pri95] A. A. B. Pritzker. *Introduction to Simulation and SLAM II.* Wiley, New York, 1995.

[RML94] V. N. Reddy, M. L. Mavrovouniotis, and M. N. Liebman. Modeling biological pathways: A discrete event systems approach. *ACS Sym. Ser.*, 576:221–234, 1994.

[RV93] A. C. Rutges and R. W. Vens. Object-oriented simulation of ecological systems using the beehive simulator. In *Proceedings of the Object Oriented Simulation Conference 1993 (OOS '93)*, pages 157–162. IEEE Computer Society, 1993.

[Tec98] Technomatix Corp. SIMPLE++ release 5.0, 1998. URL http://www.aesop.de.

[The98] The MathWorks Inc. MATLAB 5.0/SIMULINK 2.2, 1998. URL http://www.mathworks.com/products/simulink/.

[vR97] G. van Rossum. Python language and library reference manuals, 1998. URL http://www.python.org/doc/

[YFD97] T.-Y. B. Yang, G. Furnish, and P. F. Dubois. Steering object-oriented scientific computations. In *Technology of object-oriented languages and systems TOOLS 23*, pages 112–119. 1997.