Python for Scientific Computing

By itself, Python is an excellent "steering" language for scientific codes written in other languages. However, with additional basic tools, Python transforms into a high-level language suited for scientific and engineering code that's often fast enough to be immediately useful but also flexible enough to be sped up with additional extensions.

Python is an interpreted language with expressive syntax that some have compared to executable pseudocode. This might be part of the reason why I fell in love with the language in 1996, when I was seeking a way to prototype algorithms on very large data sets that overwhelmed the capabilities of the other interpreted computing environments I was familiar with. My enjoyment of programming with Python increased as I quickly learned to express complicated ideas in the syntax and objects available with it.

The idea that coding in a high-level language can greatly enhance productivity isn't new. Many scientists and engineers are typically exposed to one or more interpreted scientific computing environments early in their careers because they help them write nontrivial computational programs without getting too bogged down in syntax and compilation time lags. Python can be used in exactly this way, but its unique features offer an environment that makes it a better choice for scientists and engineers seeking a high-level language for writing scientific applications. In the rest of this special issue's articles, you'll find a feast of reasons why Python excels as a platform for scientific computing. As a small appetizer, consider this list of general features:

- A liberal open source license lets you sell, use, or distribute your Python-based application as you see fit—no extra permission necessary.
- The fact that Python runs on so many platforms means you don't have to worry about writing an application with limited portability, which also helps avoid vendor lock-in.
- The language's clean syntax yet sophisticated constructs let you write in either a procedural or fully object-oriented fashion, as the situation dictates.
- A powerful interactive interpreter allows real-time code development and live experimentation, thus eliminating the time-consuming and productivity-eating compile step from the code-then-test development process.
- The ability to extend Python with your own compiled code means that Python can be taught to do anything as fast as your hardware will allow.
- You can embed Python into an existing application, which means you can instantly add an easy-to-use veneer on top of an older, trusted application.
- The ability to interact with a wide variety of other software on your system helps you leverage the software skills you've already acquired.

TRAVIS E. OLIPHANT
Brigham Young University

- Its large number of library modules (both installed in the standard library and through additional downloads) means you can quickly construct sophisticated programs such as a Web server that solves partial differential equations, a distributed image-processing library with automatic load-balancing, or an application that collects data from the Internet, posts results to a database, and emails you with progress reports.
- The existence of Python bindings to all standard GUI toolkits means you can apply rapid development techniques when building a user interface.
- The Python community is famous for delivering quick, useful responses to user inquiries on the mailing lists, newsgroups, and IRC channels devoted to Python.
- A repository of categorized Python modules is available at www.python.org/pypi along with easy-to-install "eggs" that simplify software management.

With this small taste of Python's usefulness, let's dive in and try to uncover a little bit about the language itself. This overview is necessarily terse. Because Python is a general-purpose programming language, a wealth of additional information is available in several books and on Internet sites dedicated to it (see the "Useful References" sidebar). Python's `help` function also provides additional information.

Clean Syntax

A significant factor in Python's utility as a computing language for scientists and engineers is its clear syntax, which can make code easy to understand and maintain. Some of this syntax's specifics include code blocks defined by indentation, extensive use of namespaces (modules), easy-to-read looping constructs (including list comprehensions), exception handling, and documentation strings. Consider the following code, which computes the `sinc` function on a list of inputs:

```
from math import sin, pi
def sinc(x):
  '''Compute the sinc function:
    sin(pi*x)/(pi*x)'''
  try:
    val = (x*pi)
    return sin(val)/val
  except ZeroDivisionError:
    return 1.0
output = [sinc(x) for x in input]
```

With only a little explanation, this code is com-

USEFUL REFERENCES

- D. Ascher et al., *Numerical Python*, tech. report UCRL-MA-128569, Lawrence Livermore Nat'l Lab., 2001; http://numpy.scipy.org.
- P.F. Dubois, K. Hinsen, and J. Hugunin, "Numerical Python," *Computers in Physics*, vol. 10, no. 3, 1996, pp. 262–267.
- E. Jones et al., "SciPy: Open Source Scientific Tools for Python," 2001; www.scipy.org.
- T.E. Oliphant, *Guide to NumPy*, Trelgol, 2006; www.trelgol.org.
- M. Pilgrim, *Dive into Python*, 2004; www.diveintopython.org.
- G. van Rossum and F.L. Drake, eds., *Python Reference Manual*, Python Software Foundation, 2006; http://docs.python.org/ref/ref.html.

pletely understandable. It shows the use of namespaces because the `sin` function and the `pi` constant are kept in a separate module and must be "imported" to be used. (Alternatively, we could have used `import math` along with `math.sin` and `math.pi` to refer to the objects). Modules are either files (with a ".py" suffix) with Python code or compiled shared libraries specifically built for import into Python. Each module contains a namespace that allows grouping of similar functions, classes, and variables in a manner that lets the code scale from simple scripts to complicated applications.

This code also demonstrates exception handling: the `try/except` syntax allows for separate handling when division fails. Notice, too, that the code block defining the function is indented, but otherwise offers no indication of the code's beginning or end. This feature ensures that both the computer and the human reader have the same concept of code level without spurious characters taking up precious onscreen real estate.

The string in the code immediately after the function definition is the function *docstring*, which is useful when generating help messages or documenting functions in the output of automatic documentation tools such as `pydoc` and `epydoc`. The output list is generated from the input sequence via a compact looping construct called a *list comprehension*. This readable construct looks almost like executable English because it creates a new list whose elements are the output of `sinc(x)` for every `x` in the input sequence.

One of Python's key features is dynamic typing. Notice that the type of the `sinc` function's input is never specified—all we need is for the `sin` function to work on it. In this case, the math module's `sin` function doesn't handle complex numbers, but the

`cmath` module does, so we can use it to replace the math module if desired.

Useful Built-In Objects

Everything in Python is an object of a particular kind (or type); the standard way to construct a type is with a function call. Some types, however, are constructed from built-in, simplified syntax. The built-in scalar types are integer (infinitely large), floating point (double precision), and complex (double-precision real and imaginary parts). The syntax automatically constructs the scalars, as evidenced by the following interactive interpreter session:

```
>>> type(1), type(1.0), type(1.0j),
   type('one')
(<type 'int'>, <type 'float'>, <type
   'complex'>, <type 'str'>)
```

The '>>>' marker indicates that the interpreter is ready to receive code. Separating items with a comma automatically constructs an immutable (unchangeable once constructed) sequence called a *tuple*. In this example, I've also demonstrated the creation of a string object, which is another immutable object used extensively in Python code. An additional sequence in Python is a mutable (can be altered) list. A list can contain a sequence of any Python object (including extra lists); thus, we can use lists to construct simple multidimensional arrays:

```
>>> a = [['an', 'ecletic', 3],[5.0,
   'nothing']]
>>> print a[1][0]
5.0
```

This is an efficient, quick way to store and work with small arrays. For larger arrays, however, NumPy is better suited for managing memory and speed requirements (I'll discuss NumPy in more detail later).

Python's dictionary provides a very useful container object that lets us look up a value using a key. For example,

```
>>> a = {2: 'two', 'one':1}
>>> print a['one'], a[2]
1 two
```

The object before the ':' here is the *key* and the object after the ':' is the *value*. The key must be immutable so that the lookup step happens quickly. Dictionaries are useful for storing data for later retrieval with a key, which is a common need in scientific computing—we can even construct a very simple sparse-matrix storage scheme with dictionaries.

File objects are also important for most scientific computing projects. We use the `open` command to construct such objects:

```
>>> fid = open('simple.txt','w')
>>> fid.write("This is a string written
   to the file.")
>>> fid.close()
```

This example shows how to open a file, write a simple string to it, and then close it again (we can also close it by deleting the `fid` variable). To read from a file, we use the file object's `read` method to return a string with the file's bytes. In this example, I've also demonstrated that both double (") and single (') quotes can delimit a string constant as long as we open and close a particular string with the same quote style.

Functions and Classes

Besides offering clean syntax, Python also contributes to the construction of maintainable code by separating code into logical groups such as modules, classes (new object definitions), and functions. As I mentioned earlier, a module is a collection of Python code grouped together into a single file (with the ".py" extension), and it usually contains many related functions or classes. After using the `import` command on a module, we can use dot notation to access the functions, classes, and variables defined within the module:

```
>>> import numpy as N
>>> print N.linalg.inv([[1,2],[1,3])
[[ 3., -2.],
 [-1., 1.]])
```

This code segment calls the `inv` function from the `linalg` submodule of the `numpy` module (which I renamed to the local variable `N` for the current session). The `inv` function works on any nested sequence and finds the "matrix" inverse of a two-dimensional array.

Functions are normally defined with the syntax `def funcname(...): <indented block>`. For simple functions, we can also use a `lambda` expression, which lets us write one-line (anonymous) functions. Here's a `lambda` expression that uses a recursive formula to compute a factorial:

```
f = lambda x: (x<1) or x*f(x-1)
```

This function takes one argument and returns one

object that should be the factorial for integer input. It works because the `or` operation doesn't compute the second operand if the first is true. Anonymous functions are occasionally useful, but the standard way to define functions is to use the `def` syntax:

```
def sum_and_mean(x, sumfunc=sum,
       norm=None):
  if norm is None:
    norm = len(x)
  y = sumfunc(x)
  return y, y/float(norm)
```

This function also illustrates the use of keyword arguments to define default values for optional function arguments, as well as the use of a tuple to return more than one result from the function. The resulting tuple can be unpacked when the function is called, giving the appearance that the function returns more than one argument:

```
tot, ave = sum_and_mean([1, 4, 10, 3.0])
```

Python fits the brain of many different kinds of people partly because it supports both procedural and object-oriented programming styles. In each module, you can define functions to implement behavior or create new objects by defining classes. These new objects can have methods and attributes, including special methods that teach Python how to interpret object-specific syntax (such as infix operators). Here's a simple class that inherits from the `list` object but redefines the '+', '-', and '*' infix operators to represent element-by-element addition, subtraction, and multiplication:

```
class vector(list):
  def __add__(self, other):
    res = [x+y for x,y in zip(self,
                    other)]
    return vector(res)
  def __sub__(self, other):
    res = [x-y for x,y in zip(self,
        other)]
    return vector(res)
  def __mul__(self, other):
    res = [x*y for x,y in zip(self,
        other)]
    return vector(res)
  def tolist(self):
    return list(self)
```

This class uses list comprehension (inline looping) and the `zip` built-in function to iterate over all elements of the two inputs that perform the re-

quested operation on each input pair. The result is a list from the list-comprehension syntax, which is converted to a vector before being returned:

```
>>> v = vector([1,2,3])
>>> print v+v
[2, 4, 6]
>>> print v*v
[1, 4, 9]
>>> print v-[3,2,1]
[-2, 0, 2]
```

Classes contain attributes accessed via dot notation (`object.attribute`), and methods are attributes that can be called like a function. They're defined inside a class block to take the object itself as the first argument. This object isn't needed when calling the method on an instance of the class, so to call the `tolist()` method of a vector class's object (which is bound to the name "v"), you would write `v.tolist()`.

Special methods are prefixed and postfixed with the characters "__" to indicate that the Python interpreter automatically calls them at special times—for example, v+v is translated by the interpreter to `v.__add__(v)`.

Standard Library

Another reason why Python is so useful is that it comes prepackaged with a wealth of general-purpose libraries, popularizing the notion that it comes with "batteries included." Let's review a few modules you might want to use in your own computing projects:

- `re`, a powerful regular expression matching that significantly enhances Python's already powerful string-processing capabilities;
- `datetime`, date–time objects and tools for their manipulation;
- `decimal`, support for real numbers with user-settable precision;
- `random`, functions for random-number generation;
- `pickle`, portable serialized (stringified) representations of Python objects (let's you save Python objects to disk and load them on a different system);
- `email`, routines for parsing, handling, and generating email messages;
- `csv`, assistance for automatically reading and writing files with comma-separated values;
- `gzip`, `zlib`, and `bz2`, functions for reading and writing compressed files;
- `zipfile` and `tarfile`, functions for extracting and creating file containers;

- `mmap`, an object allowing the usage of memory-mapped files;
- `urllib`, routines for opening arbitrary URLs;
- `ctypes`, a foreign-function interface that lets Python open a shared library and directly call its exported functions;
- `os`, a cross-platform interface to the functionality that most operating systems provide; and
- `sys`, objects maintained by the interpreter that interact strongly with it (such as the module-search path and the command-line arguments passed to the program).

All these libraries let you use Python right out of the box for almost all programming projects. Besides my scientific uses of Python, for example, I've also been able to use it to manage an email list, run a Web server, and create nice-looking personalized birthday calendars for my family.

Ease of Extension

Although Python is often fast enough for many calculation needs, multidimensional `for` loops will still leave you wanting some way to speed up the code. Fortunately, Python is easily extended with a large C-API for calling Python functionality from C, connecting to non-Python compiled code, and extending the language itself by creating new Python types (or classes) in C (or C++).

An extension module is a shared library that completely mimics a Python module with variable, function, and class attributes. It's created with a single entry-point function, which Python calls when the module is imported. This entry point sets up the module by adding constants, any new defined types, and the module function table, which translates between module function names and actual C functions to call.

Developers have created many automated tools over the years to make the process of writing Python modules that use compiled code almost trivial—f2py (http://cens.ioc.ee/projects/f2py2e/), for example, allows automated calling of Fortran code; weave (www.scipy.org/Weave) allows calling of C/C++ code; Boost.Python (www.boost.org/libs/python/doc/) allows seamless integration of C++ code into Python; and Pyrex (www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/) allows the writing of an extension module in a Python-like language.

However, what really makes Python excel as a language for scientists and engineers is the NumPy extension (which you download separately). It supplies Python with a multidimensional array object whose items are stored exactly as they would be in compiled code. The extension also provides universal functions that operate rapidly over the multidimensional array on an element-by-element basis as well as additional fast functions for calculations on the array. NumPy has its own C-API, so you can write your own extensions with it. Such extensions form the foundation of SciPy, which is a collection of toolboxes to further extend Python's scientific-computing capabilities.

NumPy

NumPy is freely available at http://numpy.scipy.org and offered under a very generous license. It grew out of an original module called Numeric (and sometimes also called `numpy`) written in 1995. Numeric established most of NumPy's features, but the way Numeric handled data types wasn't flexible enough for a group of scientists at the Space Science Telescope Institute, so they built a replacement system called Numarray that introduced significant new features. However, despite a degree of compatibility between the two array systems, developers wrote various extensions that could use only one package or the other, which created a divide in the fledgling community.

In 2005, I began the process of trying to bridge this divide by creating NumPy, which essentially took the ideas that Numarray had pioneered and ported them back to the Numeric code base, significantly enhancing Numeric in the process. In October 2006, we released version 1.0 of NumPy, which has all the features of Numeric and Numarray (including full support for both of their C-APIs) plus some additional features. This article barely scratches the surface of what NumPy provides, but a full account of the package is available at www.treglol.com.

Array Objects

First and foremost, NumPy provides a homogenous, multidimensional array of a particular data type. Although the array's main goal is to hold items of the same kind, one of its available built-in data types is called an "object." An array of it can hold an arbitrary Python object at each element, thus it's effectively a heterogeneous multidimensional array.

Data types. The data type an array can hold is quite arbitrary. NumPy's array internally supports all the fundamental C data types, including 10 different kinds of signed and unsigned integers, three kinds of floats, three kinds of complex numbers, and a Boolean type. In addition, arrays can hold strings or unicode strings, and you can even define your own data type that's equivalent to a C structure (sometimes called a *record*). Thus, for example, you can have an array whose elements consist of a 20-

byte record whose first field is a 4-byte integer, second field is a 12-character string, and last field is an array of four 1-byte unsigned integers. Consider the following example, which defines such a data type to track an array of students:

```
>>> import numpy as N
>>> dt = N.dtype([('id', 'i4'),
('name', 'S12'), ('scores', 'u1', 4)])
>>> a = N.array([(1001, 'James',
[100,98,97,60]), (1002, 'Kathy',
[100,100,85,98]), (1003, 'Michael',
[84,75, 98,100]), (1004, 'John',
[84,76,82,92])], dtype=dt)
>>> a['name']
array(['James', 'Kathy', 'Michael',
'John'], dtype='-S12')
>>> a['scores']
array([[100, 98, 97, 60],
 [100, 100, 85, 98],
 [ 84, 75, 98, 100],
 [ 84, 76, 82, 92], dtype=uint8)
```

This example shows how to extract a record array's fields as arrays of another data type. Records can even be nested so that a particular field itself contains another record data type. Record arrays are a useful way to group data and are essential if the array is constructed from a complicated memory-mapped data file.

Given the potential complexity of the array data type, it's useful to think about an array in NumPy as a collection of items consuming exactly the same number of bytes. A data type object describes each element in the array, whereas the array itself provides the information regarding the array's shape.

Attributes and methods. All arrays have several attributes and methods. Some attributes can be set to alter the array's characteristics—for example, we could reshape the previously created array of students into a 2×2 array by setting its `shape` attribute:

```
>>> print a.shape, a.ndim
(4,) 1
>>> a.shape = (2,-1)
>>> print a.shape, a.ndim
(2,2) 2
```

The –1 entry in the shape-setting tuple indicates that the second dimension's shape should be whatever is necessary to use all the array elements. In this case, the missing entry is 2.

The array's methods allow quick computation or manipulation of its elements—in our example, we can use them to *sort* the one-dimensional array of students. Using the `name` field's `argsort` method returns an array of indices that sort the array. Providing this set of indices to the `take` method creates a new 1D array in the sorted order:

```
>>> b = a.take(a['name'].argsort())
>>> print b['id']
[1001 1004 1002 1003]
```

Indexing. Another useful array feature is the ability to extract specific subregions via *indexing* by using the `a[obj]` notation on an array object There are basically four kinds of array indexing: field extraction, element selection, view-based slicing, and copy-based indexing. We've already seen an exam-

What really makes Python excel as a language for scientists and engineers is the NumPy extension (which you download separately).

ple of indexing in which `obj` indicates which array field to extract; element selection occurs when `obj` is such that we extract a single element of the array. In this case, the indexing notation returns a new Python scalar object that contains the data at that location in the array. For example, let's define `a` to be an array of Gaussian random numbers and extract a particular number from it:

```
>>> import numpy as N
>>> a = N.random.randn(50,25)
>>> print a.shape, a[10,15]
(50, 25) 0.5295135653
```

View-based indexing occurs when `obj` is a slice object or a tuple of slice objects. In this case, the indexing notation returns a new array object that points to the same data as the original array. This is an important optimization that saves unnecessary memory copying but must be mentally noted to avoid unexpected data alteration. Given the previously defined array, for example,

```
>>> b=a[10:15:2, 8:13:2]; b
array([[ 0.35238367, -0.40288084,
0.10110947],
 [-0.91742114, 1.13308636, 0.00602061],
 [-0.57394525, -2.00959791, -0.3262831
]])
```

The notation `10:15:2` inside the indexing brackets tells Python to begin at element 10, end before element 15, and get every two elements—in other words, to extract elements 10, 12, and 14 from the first dimension of the array. Indexing along one dimension of a 2D array extracts a 1D array. As a result, the second range indicates that for each 1D array, we should take elements 8, 10, and 12 to form the output array. Remember this output array is just a view of the underlying data—changing elements of `b` will change the newly formed array as well.

Copy-based indexing always returns a copy of the data and occurs when `obj` is an array (or a tuple containing arrays) of a Boolean or integer data type. Boolean indexing allows masked selection of all the array's elements, which is particularly useful for setting certain elements to a particular value. The code `a[a > 0.1] -= 0.1`, for example, will decrease every element of `a` that's larger than 0.1 by 0.1. This works because arrays redefine subtraction to perform element-by-element subtraction and comparison operators to return Boolean arrays with the comparison implemented element by element.

When the indexing array has an integer data type, it's used to extract an array of specific entries. For fully specified index arrays, the returned array's shape is the same as the shape of the input indexing arrays—for example,

```
>>> b=a[ [10,12,14], [13,15,17] ]; b
array([ 1.55922631, 0.93609952,
-0.10149853])
```

Notice that the returned 1D array has elements `a[10,13]`, `a[12,15]`, and `a[14,17]` and not the cross-product array, as some would expect. We can get the cross-product by using either `b=a[ [[10],[12],[14]], [[13,15,17]] ]` or `b=a[N.ix_([10,12,14], [13,15,17])]`.

Universal Functions

Exceptional opportunities for array manipulation and extraction are only part of what makes NumPy useful for scientific computing. It also provides universal function objects (`ufuncs`), which make it simple to define functions that take N inputs and return M outputs by performing some underlying function, element by element. The basic mathematical operations of arrays are all implemented with universal functions that take two inputs and return one output—for example, when either `b` or `c` is an array object, `a=b+c` is equivalent to `a=N.add(b,c)`.

More than 50 mathematical functions are defined in the `numpy` module as universal functions, but it's easy to define your own universal functions either in compiled code (for very fast `ufuncs`) or based on a Python function (which will have `ufuncs` features but will operate more slowly). Let's look more closely at these features.

Broadcasting. `ufuncs` operate element by element, which seems to imply that all input arrays must have the same shape. Fortunately, NumPy provides a concept known as *broadcasting*, which is a specific method for arrays that don't have the same shape to try and act as if they do. Broadcasting has two rules in the form of steps:

- Make sure all arrays have the same number of dimensions by pre-pending a `1` to the shape attribute of arrays whose number of dimensions is too small. Thus, if an array of shape `(2,5)` and an array of shape `(5,)` were input into a `ufunc`, the `(5,)`-shaped array would be interpreted as a `(1,5)`-shaped array.
- Interpret the length of any axis whose length is `1` as if it were the size of the other non-unit-length arrays in the operation. Thus, if we use an array of shape `(3,6)` and an array of shape `(6,)`, the second array would first be interpreted as a `(1,6)`-shaped array and then as a `(3,6)`-shaped array.

If applying these rules fails to produce arrays of exactly the same shape, an error occurs because element-by-element operation isn't defined. As an example of `ufunc` behavior, consider the following code, which computes the outer product of two 1D arrays:

```
>>> a,b = N.array([[1,2,3],[10,20,30]])
>>> c = a[:,N.newaxis] * b; print c
[[10, 20, 30],
 [20, 40, 60],
 [30, 60, 90]]
```

The first line uses Python's ability to map a sequence object to multiple objects in one line, which is equivalent to `temp=N.array(...)` followed by `a=temp[0]` and `b=temp[1]`. This produces two 1D arrays of shape `(3,)`. The indexing manipulation in the next line selects all the elements of `a` and adds a new axis to its end, making a `(3,1)`-shaped array. Broadcasting then interprets the multiplication of a `(3,)`-shaped array as multiplication by a `(1,3)`-shaped array that produces a `(3,3)`-shaped result, which is $c_{ij} = a_i b_j$ in index notation. Note that broadcasting never copies any data to perform its

dimension upgrading—rather, copying is handled by reusing the repeated values internally.

Output arrays. All `ufuncs` take optional arguments for output arrays. Sometimes, to speed up calculations, you might want the `ufunc` to place its result in an already allocated array rather than have a fresh new allocation occur. This is especially true in a complicated calculation that has many temporaries—for example, the code `a=(b+4)*c*d` involves the creation of two strictly unnecessary temporary arrays to hold the results of intermediate calculations. If these arrays are large, we can save the significant overhead of creating and deleting the temporary arrays by writing this as

```
a=b+4
N.multiply(a, c, a)
N.multiply(a, d, a)
```

Clearly, this isn't as easy to read, but it could be essential for large simulations.

Memory-saving type conversion. When `ufuncs` are created, they must specify the data types of the required inputs and outputs. A typical `ufunc` usually has several low-level routines registered to match specific signatures. If the pattern of inputs doesn't match one of the internally supported signatures, the `ufunc` machinery will upcast any inputs (in size-limited chunks) as needed to match an available signature so the calculation can proceed. This same chunked-casting occurs if an output array is provided that isn't the same type the calculation produces.

Suppose `a` is an array of 4-byte integers, but `b` is an array of 4-byte floats, and you want to add them together. The `add ufunc` has 18 registered low-level functions for implementing the `add` operation on identical input data types that produce the same data type. For this mixed data-type operation, the `ufunc` machinery chooses 8-byte floats for the underlying operation (the "lowest" type to which we can convert both 4-byte integers and 4-byte floats without losing precision). The conversion to 8-byte floats occurs behind the scenes in chunks no larger than a user-settable buffer size. If an output array is provided, then the 8-byte result is coerced back into the output.

Array-like object wrapping. All `ufuncs` provide the ability to handle any object as an input. The only requirement is that it be converted to an array (because it has an `__array__` method, defines the array interface, or is a sequence object). If the input arrays also have the special `__array_wrap__`

method defined, then that method will be called on each result of the `ufunc`. The method call's output is returned as the `ufunc`'s output. This lets user-defined objects define an `__array_wrap__` method that takes an input array and returns the user-defined object and have those objects passed seamlessly through NumPy's `ufuncs`.

Hardware error handling. On platforms that support it, NumPy lets you query the result of hardware error flags during the computation of any `ufunc` and issue a warning, issue an error, or call a user-provided function. Hardware error flags include underflow, overflow, divide-by-zero, and invalid result. By default, all errors either give a warning or are ignored, but the `seterr` function lets you alter that behavior:

```
>>> array([1,2])/0. #  emits a warning
>>> old = N.seterr(divide='raise')
>>> array([1,2])/0. # now it will raise
an error
>>> N.seterr(**old)
```

Methods. All `ufuncs` that take two inputs and return one output can also use the `ufunc` methods `reduce`, `accumulate`, and `reduceat`, which, respectively

- perform the function repeatedly using successive elements along a particular dimension of the array and then store the result in an output variable;
- perform the function repeatedly using successive elements along a particular dimension of the array and then store each intermediate result; and
- perform the function repeatedly using successive elements along specific portions of a particular dimension of the array.

These methods admit simple interpretations for well-known operations—for example, the `sum` method of an array that sums all the elements along a particular axis is implemented using `add.reduce`. Similarly, the array's `prod` method is implemented with `multiply.reduce`.

Basic Libraries
In addition to the fundamental array object and the many implemented `ufuncs`, NumPy comes with several standard subclasses, a masked-array implementation, a simple polynomial class, set operations on 1D arrays, and a host of other useful functions. Routines for implementing Fourier transforms, basic linear algebra operations, and random-number generation are also available. Let's

look at these features more closely, starting with the provided objects:

- `matrix` is an array subclass that works with 2D arrays and redefines the '`*`' operator to be matrix-multiplication and the '`**`' operator to be matrix-power;
- `memmap` is an array subclass in which the memory is a memory-mapped file;
- `recarray` is an array subclass that allows field access by using attribute lookup and returning arrays of strings as `chararrays`;
- `chararray` is an array of strings that provide standard string methods (which operate element-by-element) as additional array methods; and
- `ma` is an additional array type called a *masked array*, which stores a mask along with each array to ignore specific values during computation.

Many functions exist for creating, manipulating, and operating on arrays, but let's focus here on a small sampling of commonly used functions:

- `convolve` curls together two 1D input sequences;
- `diag` constructs a 2D array from its diagonal or extract a diagonal from a 2D array;
- `histogram` creates a histogram from data;
- `choose` constructs an array using a choice array and additional inputs;
- `dot` sums over the last dimension of the first argument and the second-to-last dimension of the second (this extension of matrix-multiplication exploits system libraries if available at compile time);
- `empty`, `zeros`, and `ones` create arrays of a certain shape and data type filled with nothing, 0, and 1, respectively;
- `fromfunction` creates an `N-d` array from a function with N inputs, which are assumed to be index positions in the array;
- `vectorize` takes a callable object (such as a function or a method) that operates on scalar inputs and return a callable object that works on arbitrary array inputs element-by-element; and
- `lexsort` returns an array of indices showing how to sort a sequence of arguments (these arguments must all be arrays of the same shape).

The Fourier transform is an essential tool to many algorithms for arbitrary-dimensioned data. The `numpy.fft` package has the following routines:

- `fft` and `ifft`, 1D fast Fourier transform and inverse (uses $1/N$ normalization on inverse);
- `fft2` and `ifft2`, 2D fast Fourier transform and inverse; and

- `fftn` and `ifftn`, ND fast Fourier transform and inverse.

Additional routines specialize for real-valued data—for example, `rfft` and `irfft` are real-valued Fourier transforms that take real-valued inputs and return nonredundant complex-valued outputs by taking 1D Fourier transforms along a specified dimension. All the `fft` routines take N-dimensional arrays, can pad to specified lengths, and operate only along the axes specified.

Linear algebra routines (they're all under the `numpy.linalg` namespace) are built against a default basic linear algebra system (BLAS) and a stripped-down linear algebra package (LAPack) implementation (vendor-specific linear algebra libraries can be used if provided at build-time). Among the numerical routines available are those for finding a matrix inverse (`inv`), solving a system of linear equations (`solve`), finding the determinant of a matrix (`det`), finding eigenvalues and eigenvectors (`eig`), and finding the pseudo inverse of a matrix (`pinv`). Routines for the Cholesky (`cholesky`), QR (`qr`), and SVD (`svd`) decompositions are also available.

Random-number generation is an important part of most scientific computing exercises, so NumPy comes equipped with a substantial list of fast random-number generators of both continuous and discrete type. All these random-number generators reside in the `numpy.random` namespace, allow for array inputs, and produce array outputs. Each random-number generator also takes a `size=keyword` argument that specifies the shape of the output array to be created. Two convenience functions, `rand` and `randn`, produce standard uniform and standard normal random numbers using their input arguments to determine the output shape:

```
>>> a = N.random.rand(5,10,20); print
a.shape, a.std()
(5, 10, 20) 0.289855415313
>>> b = N.random.randn(6,12,22); print
b.shape, b.var()
(6, 12, 22) 1.01300101504
```

Additional random-number generators are available for producing variates from roughly 50 different distributions; some of the continuous distributions include exponential, chi-square, Gumbel, multivariate normal, noncentral F, triangular, and gamma, to name a few. The discrete distributions available include binomial, geometric, hypergeometric, Poisson, and multinomial.

f2py

As I mentioned earlier, NumPy comes installed with a powerful tool called f2py, which can parse Fortran files and construct an extension module that contains all the subroutines and functions in those files as methods. Suppose I have a file called `example.f` that has two simple Fortran routines for subtracting the element of one array from another in two different precisions:

```
      SUBROUTINE DSUB(A,B,C,N)
      DOUBLE PRECISION A(N)
      DOUBLE PRECISION B(N)
      DOUBLE PRECISION C(N)
      INTEGER N
CF2PY INTENT(OUT) :: C
CF2PY INTENT(HIDE) :: N
      DO 20 J = 1, N
         C(J) = A(J) + B(J)
      20 CONTINUE
```

With equivalent code called `SSUB` that uses `REAL` instead of `DOUBLE PRECISION`, I can make and compile an extension module (called `example`) that contains two functions (`ssub` and `dsub`). All I need to do is run

```
f2py -m example -c example.f
```

The `CF2PY` directives in the code (which are interpreted as Fortran comments) make the interface to both routines receive two input arguments and return one output argument. The argument providing the array size is hidden and passed along automatically:

```
>>> import example
>>> example.dadd([1,2,3],[4,5,6])
array([5., 7., 9.])
>>> example.sadd([1,2,3],[4,5,6])
array([5., 7., 9.], dtype=float32)
```

Notice that f2py converts input arrays and returns the output in the expected precision.

SciPy

Quite a bit of calculation and computational ability exists with just Python and the NumPy package installed, but if you're accustomed to other computational environments, you might notice a few missing tools, such as those for optimization, special functions, and image processing. SciPy builds on top of NumPy to provide such advanced tools. To do this, SciPy resurrects quite a bit of the well-tested code available at public-domain repositories such as netlib:

- The *input/output* (`io`) subpackage provides raw routines for reading and writing binary files as well as simplified routines for reading and writing files for popular data formats.
- The *linear algebra* (`linalg`) subpackage provides extended interfaces to the BLAS and LAPack libraries and has additional decompositions such as LU (`lu`) and Schur (`schur`), as well as a selection of matrix functions such as matrix exponential (`expm`), matrix square root (`sqrtm`), and matrix logarithm (`logm`).
- The *statistics* (`stats`) subpackage provides a wide variety of distribution objects for not only creating random variates but also evaluating the `pdf`, `cdf`, and inverse `cdf` of many continuous and discrete distributions.
- The *optimization* (`optimize`) subpackage provides a collection of constrained and unconstrained multivariate optimizers and function solvers.
- The *integration* (`integrate`) subpackage provides tools for integrating both functions and ordinary differential equations, including a general-purpose integrator (`quad`), a Gaussian quadrature integrator (`quadrature`), and a method that uses Romberg interpolation (`romberg`).
- The *interpolation* (`interpolate`) subpackage includes cubic splines and linear interpolation in several dimensions.
- *Weave* (`weave`) is a very useful module for calling inline C code from Python, but it's also helpful for building extension modules (by just writing the actual C code that implements the functionality).
- The *Fourier transforms* (`fftpack`) subpackage provides Fourier transforms implemented using a different wrapper to the `fftpack` library for single and double precision as well as Hilbert and inverse Hilbert transforms.
- The *special functions* (`special`) subpackage provides more than 250 special-function calculation engines, most of which are available as universal functions (`ufuncs`).
- The *sparse* (`sparse`, `linsolve`) subpackage provides sparse matrices in several different storage schemes as well as direct and iterative solution schemes.
- The *Nd-image* (`ndimage`) subpackage provides a large collection of image- and array-processing capabilities for *N*-dimensional arrays, including fast B-spline interpolation, morphology, and various filtering operations.
- The *signals and systems* (`signal`) subpackage provides routines for signal and image processing, including *N*-dimensional convolution,

fast B-spline functions, order filtering, median filtering 1D finite impulse response (FIR) and infinite impulse response linear filtering, filter design techniques, waveform generation, and various linear time invariant (LTI) system functions.

- The *maximum entropy models* (`maxentropy`) subpackage contains two classes for fitting maximum entropy models subject to linear constraints on the expectations of arbitrary feature statistics (one class is for small discrete sample spaces, whereas the other is for sample spaces that are too large to sum over).
- The *clustering* (`cluster`) subpackage contains an implementation of the K-means clustering algorithm for generating a smaller set of observations that best fit a larger set.

Naturally, this list covers only the bare bones of what SciPy's subpackages can do—you can find more information at www.scipy.org. In addition, you can use Python's `help` command on the SciPy package and all of its subpackages (using `help (scipy.<name>)`) in an interactive session once `import scipy` has been executed.

Due to space constraints, I've barely explained all the features that Python provides to the practitioner of scientific computing—for example, I've only hinted at the myriad tools available for making it easy to wrap compiled code. Likewise, I haven't really discussed NumPy's extensive C-API, which helps you build extension modules. You can glean information about these and much more by going to http://numpy.scipy.org or www.scipy.org. For general Python information, visit www.python.org—it has additional tools not necessarily integrated into NumPy or SciPy for doing computational work. Hopefully, you'll investigate and see how much easier and more efficient your daily computational work will become thanks to this powerful language.

Travis E. Oliphant is an assistant professor of electrical and computer engineering at Brigham Young University. He's a principal author of both SciPy and NumPy, and his research interests include microscale impedance imaging, MRI reconstruction in inhomogeneous fields, and any biomedical inverse problem. Oliphant has a PhD in biomedical engineering from the Mayo Graduate School. Contact him at oliphant@ee.byu.edu.