Python\_Prob\_Stat\_Machine\_Learning\_Unpingco\_\_2E\_C01

José Unpingco

Python for Probability,

Statistics, and Machine

Learning

Second Edition

Chapter 1 Getting Started with Scientific Python

Python is fundamental to data science and machine learning, as well as an everexpanding list of areas including cyber-security, and web programming. The fundamental reason for Python’s widespread use is that it provides the software glue that permits easy exchange of methods and data across core routines typically written in Fortran or C.

Python es fundamental para la ciencia de datos y el **aprendizaje automático**, así como para una lista cada vez mayor de áreas que incluyen la **seguridad cibernética y la programación web**. La razón fundamental del uso generalizado de Python es que proporciona el pegamento de software que permite un fácil intercambio de métodos y datos a través de rutinas centrales típicamente escritas en Fortran o C.

Python is a language geared toward scientists and engineers who may not have formal software development training. It is used to prototype, design, simulate, and test without getting in the way because Python provides an inherently easy and incremental development cycle, interoperability with existing codes, access to a large base of reliable open-source codes, and a hierarchical compartmentalized design philosophy. Python is known for enhancing user productivity because it reduces the development time (i.e., time spent programming) and thereby increases program run-time.

Python es un lenguaje dirigido a científicos e ingenieros que pueden no tener una formación formal en desarrollo de software. Se utiliza para crear prototipos, diseñar, simular y probar sin estorbar porque Python proporciona un ciclo de desarrollo **inherentemente** fácil e **incremental**, **interoperabilidad** con los códigos existentes, acceso a una gran base de códigos de fuente abierta confiables y un diseño compartimentado jerárquico. filosofía. **Python es conocido por mejorar la productividad del usuario porque reduce el tiempo de desarrollo** (es decir, el tiempo dedicado a la programación) y, por lo tanto, aumenta el tiempo de ejecución del programa.

Python is an interpreted language. This means that Python codes run on a Python virtual machine that provides a layer of abstraction between the code and the platform it runs on, thus making codes portable across different platforms. For example, the same script that runs on a Windows laptop can also run on a Linux-based supercomputer or on a mobile phone. This makes programming easier because the virtual machine handles the low-level details of implementing the business logic of the script on the underlying platform.

**Python es un lenguaje interpretado.** Esto significa que los códigos de Python se ejecutan en una **máquina virtual de Python** que proporciona una capa de abstracción entre el código y la plataforma en la que se ejecuta, lo que hace que los códigos sean portátiles entre diferentes plataformas. Por ejemplo, el mismo script que se ejecuta en una computadora portátil con Windows también se puede ejecutar en una supercomputadora basada en Linux o en un teléfono móvil. Esto facilita la programación porque la máquina virtual maneja los detalles de bajo nivel de la implementación de la lógica comercial del script en la plataforma subyacente.

Python is a dynamically typed language, which means that the interpreter itself figures out the representative types (e.g., floats, integers) interactively or at run-time. This is in contrast to a language like Fortran that has compilers that study the code from beginning to end, perform many compiler-level optimizations, link intimately with the existing libraries on a specific platform, and then create an executable that is henceforth liberated from the compiler.

**Python es un lenguaje tipado dinámicamente**, lo que significa que el propio intérprete determina los tipos representativos (por ejemplo, flotantes, enteros) de forma interactiva o en tiempo de ejecución. Esto contrasta con un lenguaje como Fortran que tiene compiladores que estudian el código de principio a fin, realizan muchas optimizaciones a nivel de compilador, se vinculan íntimamente con las bibliotecas existentes en una plataforma específica y luego crean un ejecutable que de ahora en adelante se libera del código. compilador.

As you may guess, the compiler’s access to the details of the underlying platform means that it can utilize optimizations that exploit chip-specific features and cache memory. Because the virtual machine abstracts away these details, it means that the Python language does not have programmable access to these kinds of optimizations. So, where is the balance between the ease of programming the virtual machine and these key numerical optimizations that are

crucial for scientific work?

Como puede suponer, el acceso del **compilador** a los detalles de la plataforma subyacente significa que puede utilizar optimizaciones que aprovechan las características específicas del chip y la memoria caché. Debido a que la máquina virtual abstrae estos detalles, significa que el lenguaje Python no tiene acceso programable a este tipo de optimizaciones. Entonces, ¿dónde está el equilibrio entre la facilidad de programación de la máquina virtual y estas optimizaciones numéricas clave que son cruciales para el trabajo científico?

The balance comes from Python’s native ability to bind to compiled Fortran and C libraries. This means that you can send intensive computations to compiled libraries directly from the interpreter. This approach has two primary advantages. First, it gives you the fun of programming in Python, with its expressive syntax and lack of visual clutter. This is a particular boon to scientists who typically want to use software as a tool as opposed to developing software as a product.

El equilibrio proviene de la **capacidad nativa** de Python para enlazar con bibliotecas Fortran y C compiladas. Esto significa que puede enviar cálculos intensivos a bibliotecas compiladas directamente desde el intérprete. Este enfoque tiene dos ventajas principales. Primero, le brinda la diversión de programar en Python, con su **sintaxis expresiva** y la falta de desorden visual. Esta es una gran ayuda para los científicos que normalmente quieren usar el software como una herramienta en lugar de desarrollar software como un producto.

The second advantage is that you can mix-and-match different compiled libraries from diverse research areas that were not otherwise designed to work together. This works because Python makes it easy to allocate and fill memory in the interpreter, pass it as input to compiled libraries, and then recover the output back in the interpreter.

La segunda ventaja es que puede mezclar y combinar diferentes bibliotecas compiladas de diversas áreas de investigación que de otro modo no fueron diseñadas para funcionar juntas. Esto funciona porque Python facilita la asignación y el llenado de memoria en el intérprete, la pasa como entrada a las bibliotecas compiladas y luego recupera la salida en el intérprete.

Moreover, Python provides a multiplatform solution for scientific codes. As an open-source project, Python itself is available anywhere you can build it, even though it typically comes standard nowadays, as part ofmany operating systems. Thismeans that once you have written your code in Python, you can just transfer the script to another platform and run it, as long as the third-party compiled libraries are also available there.

Además, Python proporciona una **solución multiplataforma** para códigos científicos. Como proyecto de código abierto, Python en sí mismo está disponible en cualquier lugar donde pueda construirlo, aunque hoy en día normalmente viene de serie, como parte de muchos sistemas operativos. Esto significa que una vez que haya escrito su código en Python, puede simplemente transferir el script a otra plataforma y ejecutarlo, siempre que las bibliotecas compiladas de terceros también estén disponibles allí.

What if the compiled libraries are absent? Building and configuring compiled libraries across multiple systems used to be a painstaking job, but as scientific Python has matured, a wide range of libraries have now become available across all of the major platforms (i.e., Windows, MacOS, Linux, Unix) as prepackaged distributions.

¿Qué pasa si las bibliotecas compiladas están ausentes? La creación y configuración de bibliotecas compiladas en múltiples sistemas solía ser un trabajo minucioso, pero a medida que Python científico ha madurado, una amplia gama de bibliotecas ahora está disponible en todas las plataformas principales (es decir, Windows, MacOS, Linux, Unix) como preempaquetado distribuciones.

Finally, scientific Python facilitates maintainability of scientific codes because Python syntax is clean, free of semi-colon litter and other visual distractions that makes code hard to read and easy to obfuscate. Python has many built-in testing, documentation, and development tools that ease maintenance. Scientific codes are usually written by scientists unschooled in software development, so having solid software development tools built into the language itself is a particular boon.

Finalmente, Python científico facilita el mantenimiento de los códigos científicos porque la sintaxis de Python es limpia, libre de puntos y comas y otras distracciones visuales que hacen que el código sea difícil de leer y fácil de ofuscar. Python tiene muchas herramientas integradas de prueba, documentación y desarrollo que facilitan el mantenimiento. Los códigos científicos generalmente son escritos por científicos sin educación en el desarrollo de software, por lo que tener herramientas de desarrollo de software sólidas integradas en el lenguaje en sí es una gran ayuda.

**1.1 Installation and Setup**

The easiest way to get started is to download the freely available Anaconda distribution provided by Anaconda (anaconda.com), which is available for all of the major platforms. On Linux, even though most of the toolchain is available via the built-in Linux package manager, it is still better to install the Anaconda distribution because it provides its own powerful package manager (i.e., conda) that can keep track of changes in the software dependencies of the packages that it supports.

La forma más fácil de comenzar es descargar la distribución de Anaconda disponible gratuitamente proporcionada por Anaconda (anaconda.com), que está disponible para todas las plataformas principales. En Linux, aunque la mayor parte de la cadena de herramientas está disponible a través del administrador de paquetes integrado de Linux, aún es mejor instalar la distribución Anaconda porque proporciona su propio administrador de paquetes potente (es decir, conda) que puede realizar un seguimiento de los cambios en el dependencias de software de los paquetes que soporta.

Note that if you do not have administrator privileges, there is also a corresponding Miniconda distribution that does not require these privileges. Regardless of your platform, we

recommend Python version 3.6 or better. Youmay have encountered other Python variants on the web, such as IronPython (Python implemented in C#) and Jython (Python implemented in Java). In this text, we focus on the C implementation of Python (i.e., known as CPython), which is, by far, the most popular implementation.

Tenga en cuenta que si no tiene privilegios de administrador, también existe una distribución de Miniconda correspondiente que no requiere estos privilegios. Independientemente de su plataforma, nosotros recomendamos Python versión 3.6 o superior. Es posible que haya encontrado otras variantes de Python en la web, como IronPython (Python implementado en C#) y Jython (Python implementado en Java). En este texto, nos enfocamos en la implementación C de Python (es decir, conocida como CPython), que es, por mucho, la implementación más popular.

These other Python variants permit specialized, native interaction with libraries in C# or Java (respectively), which is still possible (but clunky) usingCPython. Even more Python variants exist that implement the low level machinery of Python differently for various reasons, beyond interacting with native libraries in other languages.

Estas otras variantes de Python permiten una interacción nativa especializada con bibliotecas en C# o Java (respectivamente), lo que todavía es posible (pero complicado) con CPython. Incluso existen más variantes de Python que implementan la maquinaria de bajo nivel de Python de manera diferente por varias razones, además de interactuar con bibliotecas nativas en otros idiomas.

Most notable of these is Pypy that implements a just-in-time compiler (JIT) and other powerful optimizations that can substantially speed up pure Python codes. The downside of Pypy is that its coverage of some popular scientific modules (e.g., Matplotlib, Scipy) is limited or nonexistent which means that you cannot use those modules in code meant for Pypy.

El más notable de estos es Pypy, que implementa un compilador justo a tiempo (JIT) y otras potentes optimizaciones que pueden acelerar sustancialmente los códigos Python puros. La desventaja de Pypy es que su cobertura de algunos módulos científicos populares (por ejemplo, Matplotlib, Scipy) es limitada o inexistente, lo que significa que no puede usar esos módulos en código destinado a Pypy.

If you want to install a Python module that is not available via the conda manager, the pip installer is available. This installer is the main one used outside of the scientific computing community. The key difference between the two installer is that conda implements a satisfiability solver that checks for conflicts in versions among and between installed packages. This can result in conda decreasing versions of certain packages to accommodate proposed package installation. The pip installer does not check for such conflicts checks only if the proposed package already has its dependencies installed and will install them if not or remove existing incompatible modules. The following command line uses pip to install the given Python module,

Si desea instalar un módulo de Python que no está disponible a través del administrador de conda, el instalador de pip está disponible. Este instalador es el principal utilizado fuera de la comunidad informática científica. La diferencia clave entre los dos instaladores es que conda implementa un solucionador de satisfacción que verifica los conflictos en las versiones entre los paquetes instalados. Esto puede dar lugar a versiones decrecientes de ciertos paquetes para adaptarse a la instalación del paquete propuesto. El instalador de pip no verifica tales conflictos solo verifica si el paquete propuesto ya tiene sus dependencias instaladas y las instalará si no es así o eliminará los módulos incompatibles existentes. La siguiente línea de comando usa pip para instalar el módulo Python dado,

Terminal> pip install package\_name

The pip installer will download the package you want and its dependencies and install them in the existing directory tree. This works beautifully in the case where the package in question is pure-Python, without any system-specific dependencies. Otherwise, this can be a real nightmare, especially on Windows, which lacks freely available Fortran compilers. If the module in question is a C library, one way to cope is to install the freely available Visual Studio Community Edition, which usually has enough to compile many C-codes. This platform dependency is the problem that conda was designed to solve by making the binary dependencies of the various platforms available instead of attempting to compile them. On a Windows system, if you installed Anaconda and registered it as the default Python installation (it asks during the install process), then you can use the high-quality Python wheel files on Christoph Gohlke’s laboratory site at the University of California, Irvine where he kindly makes a long list of scientific modules available.1

El instalador de pip descargará el paquete que desea y sus dependencias y los instalará en el árbol de directorios existente. Esto funciona muy bien en el caso de que el paquete en cuestión sea Python puro, sin dependencias específicas del sistema. De lo contrario, esto puede ser una verdadera pesadilla, especialmente en Windows, que carece de compiladores Fortran disponibles gratuitamente. Si el módulo en cuestión es una biblioteca C, una forma de arreglárselas es instalar Visual Studio Community Edition disponible gratuitamente, que generalmente tiene suficiente para compilar muchos códigos C. Esta dependencia de la plataforma es el problema que conda fue diseñado para resolver al hacer que las dependencias binarias de las diversas plataformas estén disponibles en lugar de intentar compilarlas. En un sistema Windows, si instaló Anaconda y la registró como la instalación predeterminada de Python (se le pregunta durante el proceso de instalación), puede usar los archivos de rueda de Python de alta calidad en el sitio del laboratorio de Christoph Gohlke en la Universidad de California, Irvine, donde él amablemente pone a disposición una larga lista de módulos científicos disponible.

Failing this, you can try the conda-forge site, which is a community-powered repository of modules that conda is capable of installing, but which are not formally supported by Anaconda. Note that conda-forge allows you to share scientific Python configurations with your remote colleagues using authentication so that you can be sure that you are downloading and running code from users you trust.

De lo contrario, puede probar el sitio conda-forge, que es un repositorio de módulos impulsado por la comunidad que conda es capaz de instalar, pero que Anaconda no admite formalmente. Tenga en cuenta que conda-forge le permite compartir configuraciones científicas de Python con sus colegas remotos mediante autenticación para que pueda estar seguro de que está descargando y ejecutando código de usuarios en los que confía.

Again, if you are on Windows, and none of the above works, then you may want to consider installing a full virtual machine solution, as provided by VMWare’s Player or Oracle’s VirtualBox (both freely available under liberal terms), or with the Windows subsystem for Linux (WSL) that is built into Windows 10. Using either of these, you can set up a Linux machine running on top of Windows, which should cure these problems entirely!

Nuevamente, si está en Windows y nada de lo anterior funciona, entonces puede considerar instalar una solución de máquina virtual completa, como la proporcionada por VMWare's Player o Oracle's VirtualBox (ambos disponibles gratuitamente en términos liberales), o con el subsistema de Windows. para Linux (WSL) que está integrado en Windows 10. Con cualquiera de estos, puede configurar una máquina Linux que se ejecute sobre Windows, ¡lo que debería solucionar estos problemas por completo!

The great part of this approach is that you can share directories between the virtual machine and the Windows system so that you don’t have to maintain duplicate data files. Anaconda Linux images are also available on the cloud by Platform as a Service (PaaS) providers like Amazon Web Services and Microsoft Azure. Note that for the vast majority of users, especially newcomers to Python, the Anaconda distribution should be more than enough on any platform. It is just worth highlighting theWindows-specific issues and associated workarounds early on. Note that there are other well-maintained scientific Python Windows installers like WinPython and PythonXY. These provide the spyder integrated development environment, which is very MATLAB-like environment for transitioning MATLAB users.

La gran parte de este enfoque es que puede compartir directorios entre la máquina virtual y el sistema Windows para que no tenga que mantener archivos de datos duplicados. Las imágenes de Anaconda Linux también están disponibles en la nube por proveedores de plataforma como servicio (PaaS) como Amazon Web Services y Microsoft Azure. Tenga en cuenta que para la gran mayoría de los usuarios, especialmente los recién llegados a Python, la distribución de Anaconda debería ser más que suficiente en cualquier plataforma. Vale la pena resaltar los problemas específicos de Windows y las soluciones alternativas asociadas desde el principio. Tenga en cuenta que hay otros instaladores científicos de Python Windows bien mantenidos como WinPython y PythonXY. Estos proporcionan el entorno de desarrollo integrado de spyder, que es un entorno muy similar a MATLAB para la transición de usuarios de MATLAB.

**1.2 Numpy**

As we touched upon earlier, to use a compiled scientific library, the memory allocated in the Python interpreter must somehow reach this library as input. Furthermore, the output from these libraries must likewise return to the Python interpreter. This twoway exchange of memory is essentially the core function of the Numpy (numerical arrays in Python) module.

—--------------------------------------------------

Como mencionamos anteriormente, para usar una biblioteca científica compilada, la memoria asignada en el intérprete de Python debe llegar de alguna manera a esta biblioteca como entrada. Además, la salida de estas bibliotecas también debe volver al intérprete de Python. Este intercambio bidireccional de memoria es esencialmente la función principal del módulo Numpy (matrices numéricas en Python).

—--------------------------------------------------

Numpy is the de facto standard for numerical arrays in Python. It arose as an effort by Travis Oliphant and others to unify the preexisting numerical arrays in Python. In this section, we provide an overview and some tips for using Numpy effectively, but for much more detail, Travis’ freely available book [1] is a great place to start.

—--------------------------------------------------

Numpy es el estándar de facto para matrices numéricas en Python. Surgió como un esfuerzo de Travis Oliphant y otros para unificar las matrices numéricas preexistentes en Python. En esta sección, brindamos una descripción general y algunos consejos para usar Numpy de manera efectiva, pero para obtener más detalles, el libro disponible gratuitamente de Travis [1] es un excelente lugar para comenzar.

—--------------------------------------------------

Numpy provides specification of byte-sized arrays in Python. For example, below we create an array of three numbers, each of 4 bytes long (32-bits at 8-bits per byte) as shown by the itemsize property. The first line imports Numpy as np, which is the recommended convention. The next line creates an array of 32-bit floating-point numbers. The itemize property shows the number of bytes per item.

Numpy proporciona la especificación de matrices de tamaño de byte en Python. Por ejemplo, a continuación creamos una matriz de tres números, cada uno de 4 bytes de longitud (32 bits a 8 bits por byte), como se muestra en la propiedad itemsize. La primera línea importa Numpy como np, que es la convención recomendada. La siguiente línea crea una matriz de números de coma flotante de 32 bits. La propiedad itemize muestra el número de bytes por elemento.

>>> import numpy as np # recommended convention

>>> x = np.array([1,2,3],dtype=np.float32)

>>> x

array([1., 2., 3.], dtype=float32)

>>> x.itemsize

4

In addition to providing uniform containers for numbers, Numpy provides a comprehensive set of universal functions (i.e., ufuncs) that process arrays element-wise without additional looping semantics. Below, we show how to compute the elementwise sine using Numpy,

Además de proporcionar contenedores uniformes para números, Numpy proporciona un conjunto integral de funciones universales (es decir, ufuncs) que procesan matrices por elementos sin semántica de bucle adicional. A continuación, mostramos cómo calcular el seno por elementos usando Numpy,

>>> np.sin(np.array([1,2,3],dtype=np.float32) )

array([0.84147096, 0.9092974 , 0.14112 ], dtype=float32)

This computes the sine of the input array [1,2,3], using Numpy’s unary function, np.sin. There is another sine function in the built-in math module, but the Numpy version is faster because it does not require explicit looping (i.e., using a for loop) over each of the elements in the array. That looping happens in the compiled np.sin function itself. Otherwise,wewould have to do looping explicitly as in the following:

Esto calcula el seno de la matriz de entrada [1,2,3], usando la función unaria de Numpy, np.sin. Hay otra función sinusoidal en el módulo matemático incorporado, pero la versión de Numpy es más rápida porque no requiere bucles explícitos (es decir, usar un bucle for) sobre cada uno de los elementos de la matriz. Ese bucle ocurre en la propia función np.sin compilada. De lo contrario, tendríamos que hacer bucles explícitamente como se muestra a continuación:

>>> from math import sin

>>> [sin(i) for i in [1,2,3]] # list comprehension

[0.8414709848078965, 0.9092974268256817, 0.1411200080598672]

Numpy uses common-sense casting rules to resolve the output types. For example, if the inputs had been an integer-type, the output would still have been a floatingpoint type. In this example, we provided a Numpy array as input to the sine function. We could have also used a plain Python list instead and Numpy would have built the intermediate Numpy array (e.g., np.sin([1,1,1])). The Numpy documentation provides a comprehensive (and very long) list of available ufuncs. Numpy arrays come in many dimensions. For example, the following shows a two-dimensional 2x3 array constructed from two conforming Python lists.

Numpy usa reglas de conversión de sentido común para resolver los tipos de salida. Por ejemplo, si las entradas hubieran sido de tipo entero, la salida aún habría sido de tipo punto flotante. En este ejemplo, proporcionamos una matriz Numpy como entrada para la función seno. En su lugar, también podríamos haber usado una lista simple de Python y Numpy habría creado la matriz intermedia de Numpy (por ejemplo, np.sin([1,1,1])). La documentación de Numpy proporciona una lista completa (y muy larga) de ufuncs disponibles. Las matrices Numpy vienen en muchas dimensiones. Por ejemplo, lo siguiente muestra una matriz bidimensional de 2x3 construida a partir de dos listas de Python conformes.

>>> x=np.array([ [1,2,3],[4,5,6] ])

>>> x.shape

(2, 3)

Note that Numpy is limited to 32 dimensions unless you build it for more.2 Numpy arrays follow the usual Python slicing rules in multiple dimensions as shown below where the : colon character selects all elements along a particular axis.

Tenga en cuenta que Numpy está limitado a 32 dimensiones a menos que lo construya para más.2 Las matrices Numpy siguen las reglas habituales de división de Python en múltiples dimensiones, como se muestra a continuación, donde el carácter : dos puntos selecciona todos los elementos a lo largo de un eje particular.

>>> x=np.array([ [1,2,3],[4,5,6] ])

>>> x[:,0] # 0th column

array([1, 4])

>>> x[:,1] # 1st column

array([2, 5])

>>> x[0,:] # 0th row

array([1, 2, 3])

>>> x[1,:] # 1st row

array([4, 5, 6])

You can also select subsections of arrays by using slicing as shown below

>>> x=np.array([ [1,2,3],[4,5,6] ])

>>> x

array([[1, 2, 3],

[4, 5, 6]])

>>> x[:,1:] # all rows, 1st thru last column

array([[2, 3],

[5, 6]])

>>> x[:,::2] # all rows, every other column

array([[1, 3],

[4, 6]])

>>> x[:,::-1] # reverse order of columns

array([[3, 2, 1],

[6, 5, 4]])

**1.2.1 Numpy Arrays and Memory**

Some interpreted languages implicitly allocate memory. For example, in MATLAB, you can extend a matrix by simply tacking on another dimension as in the following MATLAB session:

Algunos lenguajes interpretados asignan memoria implícitamente. Por ejemplo, en MATLAB, puede extender una matriz simplemente agregando otra dimensión como en la siguiente sesión de MATLAB:

>> x=ones(3,3)

x =

1 1 1

1 1 1

1 1 1

>> x(:,4)=ones(3,1) % tack on extra dimension

x =

1 1 1 1

1 1 1 1

1 1 1 1

>> size(x)

ans =

3 4

This works because MATLAB arrays use pass-by-value semantics so that slice operations actually copy parts of the array as needed. By contrast, Numpy uses pass-byreference semantics so that slice operations are views into the array without implicit copying. This is particularly helpful with large arrays that already strain available memory. In Numpy terminology, slicing creates views (no copying) and advanced indexing creates copies. Let’s start with advanced indexing. If the indexing object (i.e., the item between the brackets) is a non-tuple sequence object, another Numpy array (of type integer or boolean), or a tuple with at least one sequence object or Numpy array, then indexing creates copies. For the above example, to accomplish the same array extension inNumpy, you have to do something like the following:

Esto funciona porque los arreglos de MATLAB usan semántica de paso por valor para que las operaciones de división copie partes del arreglo según sea necesario. Por el contrario, Numpy usa la semántica de paso por referencia para que las operaciones de corte sean vistas en la matriz sin copia implícita. Esto es particularmente útil con arreglos grandes que ya agotan la memoria disponible. En la terminología de Numpy, el corte crea vistas (sin copiar) y la indexación avanzada crea copias. Comencemos con la indexación avanzada. Si el objeto de indexación (es decir, el elemento entre corchetes) es un objeto de secuencia que no es una tupla, otra matriz Numpy (de tipo entero o booleano) o una tupla con al menos un objeto de secuencia o una matriz Numpy, la indexación crea copias. Para el ejemplo anterior, para lograr la misma extensión de matriz en Numpy, debe hacer algo como lo siguiente:

>>> x = np.ones((3,3))

>>> x

array([[1., 1., 1.],

[1., 1., 1.],

[1., 1., 1.]])

>>> x[:,[0,1,2,2]] # notice duplicated last dimension

array([[1., 1., 1., 1.],

[1., 1., 1., 1.],

[1., 1., 1., 1.]])

>>> y=x[:,[0,1,2,2]] # same as above, but do assign it to y

Because of advanced indexing, the variable y has its own memory because the relevant parts of x were copied. To prove it, we assign a new element to x and see that y is not updated.

Debido a la indexación avanzada, la variable y tiene su propia memoria porque se copiaron las partes relevantes de x. Para probarlo, asignamos un nuevo elemento a x y vemos que y no se actualiza.

>>> x[0,0]=999 # change element in x

>>> x # changed

array([[999., 1., 1.],

[ 1., 1., 1.],

[ 1., 1., 1.]])

>>> y # not changed!

array([[1., 1., 1., 1.],

[1., 1., 1., 1.],

[1., 1., 1., 1.]])

However, if we start over and construct y by slicing (whichmakes it a view) as shown below, then the change we made does affect y because a view is just a window into the same memory.

Sin embargo, si comenzamos de nuevo y construimos y cortando (lo que la convierte en una vista) como se muestra a continuación, entonces el cambio que hicimos sí afecta a y porque una vista es solo una ventana en la misma memoria.

>>> x = np.ones((3,3))

>>> y = x[:2,:2] # view of upper left piece

>>> x[0,0] = 999 # change value

>>> x

array([[999., 1., 1.],

[ 1., 1., 1.],

[ 1., 1., 1.]])

>>> y

array([[999., 1.],

[ 1., 1.]])

Note that if you want to explicitly force a copy without any indexing tricks, you can do y=x.copy(). The code below works through another example of advanced indexing versus slicing.

Tenga en cuenta que si desea forzar explícitamente una copia sin ningún truco de indexación, puede hacer y=x.copy(). El siguiente código funciona a través de otro ejemplo de indexación avanzada versus división.

>>> x = np.arange(5) # create array

>>> x

array([0, 1, 2, 3, 4])

>>> y=x[[0,1,2]] # index by integer list to force copy

>>> y

array([0, 1, 2])

>>> z=x[:3] # slice creates view

>>> z # note y and z have same entries

array([0, 1, 2])

>>> x[0]=999 # change element of x

>>> x

array([999, 1, 2, 3, 4])

>>> y # note y is unaffected,

array([0, 1, 2])

>>> z # but z is (it's a view).

array([999, 1, 2])

In this example, y is a copy, not a view, because it was created using advanced indexing whereas z was created using slicing. Thus, even though y and z have the same entries, only z is affected by changes to x. Note that the flags property of Numpy arrays can help sort this out until you get used to it. Manipulating memory using views is particularly powerful for signal and image processing algorithms that require overlapping fragments of memory. The following is an example of how to use advanced Numpy to create overlapping blocks that do not actually consume additional memory,

En este ejemplo, y es una copia, no una vista, porque se creó mediante indexación avanzada, mientras que z se creó mediante división. Por lo tanto, aunque y y z tengan las mismas entradas, solo z se ve afectado por los cambios en x. Tenga en cuenta que la propiedad flags de las matrices Numpy puede ayudar a resolver esto hasta que se acostumbre. Manipular la memoria usando vistas es particularmente poderoso para los algoritmos de procesamiento de señales e imágenes que requieren fragmentos de memoria superpuestos. El siguiente es un ejemplo de cómo usar Numpy avanzado para crear bloques superpuestos que en realidad no consumen memoria adicional,

>>> from numpy.lib.stride\_tricks import as\_strided

>>> x = np.arange(16,dtype=np.int64)

>>> y=as\_strided(x,(7,4),(16,8)) # overlapped entries

>>> y

array([[ 0, 1, 2, 3],

[ 2, 3, 4, 5],

[ 4, 5, 6, 7],

[ 6, 7, 8, 9],

[ 8, 9, 10, 11],

[10, 11, 12, 13],

[12, 13, 14, 15]])

The above code creates a range of integers and then overlaps the entries to create a 7x4 Numpy array. The final argument in the as\_strided function are the strides, which are the steps in bytes to move in the row and column dimensions, respectively. Thus, the resulting array steps eight bytes in the column dimension and sixteen bytes in the row dimension. Because the integer elements in the Numpy array are eight bytes, this is equivalent to moving by one element in the column dimension and by two elements in the row dimension. The second row in the Numpy array starts at sixteen bytes (two elements) from the first entry (i.e., 2) and then proceeds by eight bytes (by one element) in the column dimension (i.e., 2,3,4,5). The important part is that memory is re-used in the resulting 7x4 Numpy array. The code below demonstrates this by reassigning elements in the original x array. The changes show up in the y array because they point at the same allocated memory.

El código anterior crea un rango de números enteros y luego superpone las entradas para crear una matriz Numpy de 7x4. El argumento final en la función as\_strided son los pasos, que son los pasos en bytes para moverse en las dimensiones de fila y columna, respectivamente. Por lo tanto, la matriz resultante avanza ocho bytes en la dimensión de columna y dieciséis bytes en la dimensión de fila. Debido a que los elementos enteros en la matriz Numpy son ocho bytes, esto es equivalente a moverse un elemento en la dimensión de la columna y dos elementos en la dimensión de la fila. La segunda fila en la matriz Numpy comienza en dieciséis bytes (dos elementos) desde la primera entrada (es decir, 2) y luego avanza por ocho bytes (por un elemento) en la dimensión de la columna (es decir, 2,3,4,5) . La parte importante es que la memoria se reutiliza en la matriz Numpy de 7x4 resultante. El siguiente código demuestra esto al reasignar elementos en la matriz x original. Los cambios se muestran en la matriz y porque apuntan a la misma memoria asignada.

>>> x[::2]=99 # assign every other value

>>> x

array([99, 1, 99, 3, 99, 5, 99, 7, 99, 9, 99, 11, 99, 13, 99, 15])

>>> y # the changes appear because y is a view

array([[99, 1, 99, 3],

[99, 3, 99, 5],

[99, 5, 99, 7],

[99, 7, 99, 9],

[99, 9, 99, 11],

[99, 11, 99, 13],

[99, 13, 99, 15]])

Bear in mind that as\_strided does not check that you stay within memory block bounds. So, if the size of the target matrix is not filled by the available data, the remaining elements will come from whatever bytes are at that memory location. In other words, there is no default filling by zeros or other strategy that defends memory block bounds. One defense is to explicitly control the dimensions as in the following code:

Tenga en cuenta que as\_strided no verifica que permanezca dentro de los límites del bloque de memoria. Por lo tanto, si el tamaño de la matriz de destino no se llena con los datos disponibles, los elementos restantes provendrán de los bytes que se encuentren en esa ubicación de memoria. En otras palabras, no existe un relleno predeterminado con ceros u otra estrategia que defienda los límites del bloque de memoria. Una defensa es controlar explícitamente las dimensiones como en el siguiente código:

>>> n = 8 # number of elements

>>> x = np.arange(n) # create array

>>> k = 5 # desired number of rows

>>> y = as\_strided(x,(k,n-k+1),(x.itemsize,)\*2)

>>> y

array([[0, 1, 2, 3],

[1, 2, 3, 4],

[2, 3, 4, 5],

[3, 4, 5, 6],

[4, 5, 6, 7]])

1.2.2 Numpy Matrices

Matrices in Numpy are similar to Numpy arrays but they can only have two dimensions. They implement row–column matrix multiplication as opposed to elementwise multiplication. If you have two matrices you want to multiply, you can either create them directly or convert them from Numpy arrays. For example, the following shows how to create two matrices and multiply them.

Las matrices en Numpy son similares a las matrices de Numpy, pero solo pueden tener dos dimensiones. Implementan la multiplicación de matrices fila-columna en lugar de la multiplicación por elementos. Si tiene dos matrices que desea multiplicar, puede crearlas directamente o convertirlas desde matrices Numpy. Por ejemplo, lo siguiente muestra cómo crear dos matrices y multiplicarlas.

>>> import numpy as np

>>> A=np.matrix([[1,2,3],[4,5,6],[7,8,9]])

>>> x=np.matrix([[1],[0],[0]])

>>> A\*x

matrix([[1],

[4],

[7]])

This can also be done using arrays as shown below

>>> A=np.array([[1,2,3],[4,5,6],[7,8,9]])

>>> x=np.array([[1],[0],[0]])

>>> A.dot(x)

array([[1],

[4],

[7]])

Numpy arrays support element-wise multiplication, not row–column multiplication. You must use Numpy matrices for this kind of multiplication unless use the inner product np.dot, which also works in multiple dimensions (see np.tensordot for more general dot products). Note that Python 3.x has a new @ notation for matrix multiplication so we can re-do the last calculation as follows:

Las matrices Numpy admiten la multiplicación por elementos, no la multiplicación por filas y columnas. Debe usar matrices Numpy para este tipo de multiplicación a menos que use el producto interno np.dot, que también funciona en múltiples dimensiones (vea np.tensordot para productos de puntos más generales). Tenga en cuenta que Python 3.x tiene una nueva notación @ para la multiplicación de matrices, por lo que podemos volver a hacer el último cálculo de la siguiente manera:

>>> A @ x

array([[1],

[4],

[7]])

It is unnecessary to cast all multiplicands to matrices for multiplication. In the next example, everything until last line is a Numpy array and thereafter we cast the array as a matrix with np.matrix which then uses row–columnmultiplication. Note that it is unnecessary to cast the x variable as a matrix because the left-to-right order of the evaluation takes care of that automatically. If we need to use A as a matrix elsewhere in the code then we should bind it to another variable instead of re-casting it every time. If you find yourself casting back and forth for large arrays, passing the copy=False flag to matrix avoids the expense of making a copy.

No es necesario convertir todos los multiplicandos en matrices para la multiplicación. En el siguiente ejemplo, todo hasta la última línea es una matriz Numpy y luego convertimos la matriz como una matriz con np.matrix que luego usa la multiplicación de filas y columnas. Tenga en cuenta que no es necesario convertir la variable x en una matriz porque el orden de evaluación de izquierda a derecha se ocupa de eso automáticamente. Si necesitamos usar A como una matriz en otra parte del código, entonces deberíamos vincularla a otra variable en lugar de volver a emitirla cada vez. Si se encuentra buscando matrices grandes, pasar el indicador copy=False a matrix evita el gasto de hacer una copia.

>>> A=np.ones((3,3))

>>> type(A) # array not matrix

<class 'numpy.ndarray'>

>>> x=np.ones((3,1)) # array not matrix

>>> A\*x

array([[1., 1., 1.],

[1., 1., 1.],

[1., 1., 1.]])

>>> np.matrix(A)\*x # row-column multiplication

matrix([[3.],

[3.],

[3.]])

1.2.3 Numpy Broadcasting

Numpy broadcasting is a powerful way to make implicit multidimensional grids for expressions. It is probably the single most powerful feature of Numpy and the most difficult to grasp. Proceeding by example, consider the vertices of a two-dimensional unit square as shown below

La transmisión numpy es una forma poderosa de crear cuadrículas multidimensionales implícitas para las expresiones. Es probablemente la característica más poderosa de Numpy y la más difícil de comprender. Procediendo con el ejemplo, considere los vértices de un cuadrado unitario bidimensional como se muestra a continuación

>>> X,Y=np.meshgrid(np.arange(2),np.arange(2))

>>> X

array([[0, 1],

[0, 1]])

>>> Y

array([[0, 0],

[1, 1]])

Numpy’s meshgrid creates two-dimensional grids. The X and Y arrays have corresponding entries match the coordinates of the vertices of the unit square (e.g., (0, 0), (0, 1), (1, 0), (1, 1)). To add the x and y-coordinates, we could use X and Y as in X+Y shown below, The output is the sum of the vertex coordinates of the unit square.

La rejilla de malla de Numpy crea rejillas bidimensionales. Las matrices X e Y tienen entradas correspondientes que coinciden con las coordenadas de los vértices del cuadrado unitario (por ejemplo, (0, 0), (0, 1), (1, 0), (1, 1)). Para sumar las coordenadas x e y, podríamos usar X e Y como en X+Y que se muestra a continuación. La salida es la suma de las coordenadas del vértice del cuadrado unitario.

>>> X+Y

array([[0, 1],

[1, 2]])

Because the two arrays have compatible shapes, they can be added together elementwise. It turns out we can skip a step here and not bother with meshgrid to implicitly obtain the vertex coordinates by using broadcasting as shown below

Debido a que las dos matrices tienen formas compatibles, se pueden sumar por elementos. Resulta que podemos omitir un paso aquí y no molestarnos con meshgrid para obtener implícitamente las coordenadas del vértice mediante la transmisión como se muestra a continuación.

>>> x = np.array([0,1])

>>> y = np.array([0,1])

>>> x

array([0, 1])

>>> y

array([0, 1])

>>> x + y[:,None] # add broadcast dimension

array([[0, 1],

[1, 2]])

>>> X+Y

array([[0, 1],

[1, 2]])

On line 7 the None Python singleton tells Numpy to make copies of y along this dimension to create a conformable calculation. Note that np.newaxis can be used instead of None to be more explicit. The following lines show that we obtain the same output as when we used the X+Y Numpy arrays. Note that without broadcasting x+y=array([0, 2]) which is not what we are trying to compute. Let’s continue with a more complicated example where we have differing array shapes.

En la línea 7, el singleton Ninguno Python le dice a Numpy que haga copias de y a lo largo de esta dimensión para crear un cálculo conforme. Tenga en cuenta que se puede usar np.newaxis en lugar de None para ser más explícito. Las siguientes líneas muestran que obtenemos el mismo resultado que cuando usamos las matrices X+Y Numpy. Tenga en cuenta que sin transmitir x+y=array([0, 2]), que no es lo que estamos tratando de calcular. Continuemos con un ejemplo más complicado en el que tenemos diferentes formas de matriz.

>>> x = np.array([0,1])

>>> y = np.array([0,1,2])

>>> X,Y = np.meshgrid(x,y)

>>> X

array([[0, 1],

[0, 1],

[0, 1]])

>>> Y

array([[0, 0],

[1, 1],

[2, 2]])

>>> X+Y

array([[0, 1],

[1, 2],

[2, 3]])

>>> x+y[:,None] # same as with meshgrid

array([[0, 1],

[1, 2],

[2, 3]])

In this example, the array shapes are different, so the addition of x and y is not possible without Numpy broadcasting. The last line shows that broadcasting generates the same output as using the compatible array generated by meshgrid. This shows that broadcasting works with different array shapes. For the sake of comparison, on line 3, meshgrid creates two conformable arrays, X and Y. On the last line, x+y[:,None] produces the same output as X+Y without the meshgrid. We can also put the None dimension on the x array as x[:,None]+y which would give the transpose of the result.

En este ejemplo, las formas de la matriz son diferentes, por lo que la suma de x e y no es posible sin la transmisión de Numpy. La última línea muestra que la transmisión genera el mismo resultado que el uso de la matriz compatible generada por meshgrid. Esto muestra que la transmisión funciona con diferentes formas de matriz. En aras de la comparación, en la línea 3, meshgrid crea dos matrices compatibles, X e Y. En la última línea, x+y[:,None] produce el mismo resultado que X+Y sin meshgrid. También podemos poner la dimensión Ninguno en la matriz x como x[:,Ninguno]+y, lo que daría la transposición del resultado.

Broadcasting works in multiple dimensions also. The output shown has shape (4,3,2). On the last line, the x+y[:,None] produces a two-dimensional array which is then broadcast against z[:,None,None], which duplicates itself along the two added dimensions to accommodate the two-dimensional result on its left (i.e., x + y[:,None]). The caveat about broadcasting is that it can potentially create large, memory-consuming, intermediate arrays. There are methods for controlling this by re-using previously allocated memory but that is beyond our scope here. Formulas in physics that evaluate functions on the vertices of high dimensional grids are great use-cases for broadcasting.

La radiodifusión también funciona en múltiples dimensiones. La salida que se muestra tiene forma (4,3,2). En la última línea, x+y[:,None] produce una matriz bidimensional que luego se transmite contra z[:,None,None], que se duplica a lo largo de las dos dimensiones agregadas para acomodar el resultado bidimensional en su izquierda (es decir, x + y[:,Ninguno]). La advertencia sobre la transmisión es que potencialmente puede crear matrices intermedias grandes que consumen memoria. Existen métodos para controlar esto mediante la reutilización de la memoria previamente asignada, pero eso está más allá de nuestro alcance aquí. Las fórmulas en física que evalúan funciones en los vértices de cuadrículas de alta dimensión son excelentes casos de uso para la transmisión.

>>> x = np.array([0,1])

>>> y = np.array([0,1,2])

>>> z = np.array([0,1,2,3])

>>> x+y[:,None]+z[:,None,None]

array([[[0, 1],

[1, 2],

[2, 3]],

[[1, 2],

[2, 3],

[3, 4]],

[[2, 3],

[3, 4],

[4, 5]],

[[3, 4],

[4, 5],

[5, 6]]])

1.2.4 Numpy Masked Arrays

Numpy provides a powerful method to temporarily hide array elements without

changing the shape of the array itself,

>>> from numpy import ma # import masked arrays

>>> x = np.arange(10)

>>> y = ma.masked\_array(x, x<5)

>>> print (y)

[-- -- -- -- -- 5 6 7 8 9]

>>> print (y.shape)

(10,)

Note that the elements in the array for which the logical condition (x<5) is true are masked, but the size of the array remains the same. This is particularly useful in plotting categorical data, where you may only want those values that correspond to a given category for part of the plot. Another common use is for image processing, wherein parts of the image may need to be excluded from subsequent processing. Note that creating a masked array does not force an implicit copy operation unless copy=True argument is used. For example, changing an element in x does change the corresponding element in y, even though y is a masked array,

Tenga en cuenta que los elementos de la matriz para los que la condición lógica (x<5) es verdadera están enmascarados, pero el tamaño de la matriz sigue siendo el mismo. Esto es particularmente útil en la representación gráfica de datos categóricos, donde es posible que solo desee aquellos valores que correspondan a una categoría determinada para parte de la representación gráfica. Otro uso común es para el procesamiento de imágenes, en el que es posible que sea necesario excluir partes de la imagen del procesamiento posterior. Tenga en cuenta que la creación de una matriz enmascarada no fuerza una operación de copia implícita a menos que se use el argumento copy=True. Por ejemplo, cambiar un elemento en x cambia el elemento correspondiente en y, aunque y es una matriz enmascarada,

>>> x[-1] = 99 # change this

>>> print(x)

[ 0 1 2 3 4 5 6 7 8 99]

>>> print(y)# masked array changed!

[-- -- -- -- -- 5 6 7 8 99]

1.2.5 Floating-Point Numbers

There are precision limitations when representing floating-point numbers on a computer with finite memory. For example, the following shows these limitations when adding two simple numbers,

Existen limitaciones de precisión cuando se representan números de coma flotante en una computadora con memoria finita. Por ejemplo, lo siguiente muestra estas limitaciones al sumar dos números simples,

>>> 0.1 + 0.2

0.30000000000000004

So, then, why is the output not 0.3? The issue is the floating-point representation of the two numbers and the algorithm that adds them. To represent an integer in binary, we just write it out in powers of 2. For example, 230 = (11100110)2. Python can do this conversion using string formatting,

Entonces, ¿por qué la salida no es 0.3? El problema es la representación de punto flotante de los dos números y el algoritmo que los suma. Para representar un número entero en binario, simplemente lo escribimos en potencias de 2. Por ejemplo, 230 = (11100110)2. Python puede hacer esta conversión usando formato de cadena,

>>> print('{0:b}'.format(230))

11100110

To add integers, we just add up the corresponding bits and fit them into the allowable number of bits. Unless there is an overflow (the results cannot be represented with that number of bits), then there is no problem. Representing floating point is trickier because we have to represent these numbers as binary fractions. The IEEE 754 standard requires that floating-point numbers be represented as Å}C Å~2E where C is the significand (mantissa) and E is the exponent.

Para sumar números enteros, simplemente sumamos los bits correspondientes y los colocamos en el número de bits permitido. A menos que haya un desbordamiento (los resultados no se pueden representar con esa cantidad de bits), entonces no hay problema. Representar el punto flotante es más complicado porque tenemos que representar estos números como fracciones binarias. El estándar IEEE 754 requiere que los números de coma flotante se representen como Å}C Å~2E donde C es la mantisa y E es el exponente.

To represent a regular decimal fraction as binary fraction, we need to compute the expansion of the fraction in the following form a1/2+a2/22 +a3/23... In other words, we need to find the ai coefficients.We can do this using the same process we would use for a decimal fraction: just keep dividing by the fractional powers of ½ and keep track of the whole and fractional parts. Python’s divmod function can do most of the work for this. For example, to represent 0.125 as a binary fraction,

Para representar una fracción decimal regular como fracción binaria, necesitamos calcular la expansión de la fracción de la siguiente forma a1/2+a2/22 +a3/23... En otras palabras, necesitamos encontrar los coeficientes ai. puede hacer esto usando el mismo proceso que usaríamos para una fracción decimal: simplemente siga dividiendo por las potencias fraccionarias de ½ y lleve la cuenta de las partes enteras y fraccionarias. La función divmod de Python puede hacer la mayor parte del trabajo para esto. Por ejemplo, para representar 0.125 como una fracción binaria,

>>> a = 0.125

>>> divmod(a\*2,1)

(0.0, 0.25)

The first item in the tuple is the quotient and the other is the remainder. If the quotient was greater than 1, then the corresponding ai term is one and is zero otherwise. For this example, we have a1 = 0. To get the next term in the expansion, we just keep multiplying by 2 which moves us rightward along the expansion to ai+1 and so on. Then,

El primer elemento de la tupla es el cociente y el otro es el resto. Si el cociente fue mayor que 1, entonces el término ai correspondiente es uno y cero en caso contrario. Para este ejemplo, tenemos a1 = 0. Para obtener el siguiente término en la expansión, seguimos multiplicando por 2, lo que nos mueve hacia la derecha a lo largo de la expansión hasta ai+1 y así sucesivamente. Entonces,

>>> a = 0.125

>>> q,a = divmod(a\*2,1)

>>> print (q,a)

0.0 0.25

>>> q,a = divmod(a\*2,1)

>>> print (q,a)

0.0 0.5

>>> q,a = divmod(a\*2,1)

>>> print (q,a)

1.0 0.0

The algorithm stops when the remainder term is zero. Thus, we have that 0.125 = (0.001)2. The specification requires that the leading term in the expansion be one. Thus, we have 0.125 = (1.000) Å~ 2−3. This means the significand is 1 and the exponent is -3. Now, let’s get back to our main problem 0.1+0.2 by developing the representation 0.1 by coding up the individual steps above.

El algoritmo se detiene cuando el término restante es cero. Así, tenemos que 0.125 = (0.001)2. La especificación requiere que el término principal en la expansión sea uno. Por tanto, tenemos 0,125 = (1,000) Å~ 2−3. Esto significa que el significado es 1 y el exponente es -3. Ahora, regresemos a nuestro problema principal 0.1+0.2 desarrollando la representación 0.1 codificando los pasos individuales anteriores.

>>> a = 0.1

>>> bits = []

>>> while a>0:

... q,a = divmod(a\*2,1)

... bits.append(q)

...

>>> print (''.join(['%d'%i for i in bits]))

0001100110011001100110011001100110011001100110011001101

Note that the representation has an infinitely repeating pattern. This means that we have (1.1001)2Å~2−4. The IEEE standard does not have a way to represent infinitely repeating sequences. Nonetheless, we can compute this,

Tenga en cuenta que la representación tiene un patrón que se repite infinitamente. Esto significa que tenemos (1.1001)2Å~2−4. El estándar IEEE no tiene una forma de representar secuencias que se repiten infinitamente. No obstante, podemos calcular esto,

∞

n=1

1

24n−3

+ 1

24n

= 3

5

Thus, 0.1 ≈ 1.6 Å~ 2−4. Per the IEEE 754 standard, for float type, we have

24-bits for the significand and 23-bits for the fractional part. Because we cannot

represent the infinitely repeating sequence, we have to round off at 23-bits,

10011001100110011001101. Thus, whereas the significand’s representation used

to be 1.6, with this rounding, it is Now

>>> b = '10011001100110011001101'

>>> 1+sum([int(i)/(2\*\*n) for n,i in enumerate(b,1)])

1.600000023841858

Thus,we now have 0.1 ≈ 1.600000023841858Å~2−4 = 0.10000000149011612. For

the 0.2 expansion, we have the same repeating sequence with a different exponent,

so that we have 0.2 ≈ 1.600000023841858 Å~ 2−3 = 0.20000000298023224. To

add 0.1+0.2 in binary, we must adjust the exponents until they match the higher of

the two. Thus,

0.11001100110011001100110

+1.10011001100110011001101

--------------------------

10.01100110011001100110011

Now, the sum has to be scaled back to fit into the significand’s available bits so the

result is 1.00110011001100110011010 with exponent -2. Computing this in the

usual way as shown below gives the result.

>>> k='00110011001100110011010'

>>> print('%0.12f'%((1+sum([int(i)/(2\*\*n)

... for n,i in enumerate(k,1)]))/2\*\*2))

0.300000011921

which matches what we get with numpy

>>> import numpy as np

>>> print('%0.12f'%(np.float32(0.1) + np.float32(0.2)))

0.300000011921

The entire process proceeds the same for 64-bit floats. Python has a fractions

and decimal modules that allow more exact number representations. The decimal

module is particularly important for certain financial computations.

Round-off Error. Let’s consider the example of adding 100,000,000 and 10 in

32-bit floating point.

>>> print('{0:b}'.format(100000000))

101111101011110000100000000

This means that 100, 000, 000 = (1.01111101011110000100000000)2 Å~226. Likewise,

10 = (1.010)2 Å~ 23. To add these we have to make the exponents match as in

the following,

1.01111101011110000100000000

+0.00000000000000000000001010

-------------------------------

1.01111101011110000100001010

Now, we have to round off because we only have 23 bits to the right of the decimal

point and obtain 1.0111110101111000010000, thus losing the trailing 10 bits.

This effectively makes the decimal 10 = (1010)2 we started out with become 8 =

(1000)2. Thus, using Numpy again,

>>> print(format(np.float32(100000000) + np.float32(10),'10.3f'))

100000008.000

The problem here is that the order of magnitude between the two numbers was so

great that it resulted in loss in the significand’s bits as the smaller number was rightshifted.

When summing numbers like these, the Kahan summation algorithm (see

math.fsum()) can effectively manage these round-off errors.

>>> import math

>>> math.fsum([np.float32(100000000),np.float32(10)])

100000010.0

Cancelation Error. Cancelation error (loss of significance) results when two nearly

equal floating-point numbers are subtracted. Let’s consider subtracting 0.1111112

and 0.1111111. As binary fractions, we have the following,

1.11000111000111001000101 E-4

-1.11000111000111000110111 E-4

---------------------------

0.00000000000000000011100

As a binary fraction, this is 1.11 with exponent -23 or (1.75)10 Å~ 2−23 ≈

0.00000010430812836. In Numpy, this loss of precision is shown in the following:

>>> print(format(np.float32(0.1111112)-np.float32(0.1111111),'1.17f'))

0.00000010430812836

To sum up, when using floating point, you must check for approximate equality using

something like Numpy allclose instead of the usual Python equality (i.e., ==)

sign. This enforces error bounds instead of strict equality.Whenever practicable, use

fixed scaling to employ integer values instead of decimal fractions. Double precision

64-bit floating-point numbers are much better than single precision and, while

not eliminating these problems, effectively kicks the can down the road for all but

the strictest precision requirements. The Kahan algorithm is effective for summing

floating point numbers across very large data without accruing round-off errors. To

minimize cancelation errors, re-factor the calculation to avoid subtracting two nearly

equal numbers.

1.2.6 Numpy Optimizations and Prospectus

The scientific Python community continues to push the frontier of scientific computing.

Several important extensions to Numpy are under active development. First,

Numba is a compiler that generates optimized machine code from pure-Python code

using the LLVMcompiler infrastructure. LLVMstarted as a research project at theUniversity

of Illinois to provide a target-independent compilation strategy for arbitrary

programming languages and is now a well-established technology. The combination

of LLVMand Python via Numba means that accelerating a block of Python code can

be as easy as putting a @numba.jit decorator above the function definition, but this

doesn’t work for all situations. Numba can target general graphics processing units

(GPGPUs) also.

The Dask project contains dask.array extensions for manipulating very large

datasets that are too big to fit in a single computer’s RAM (i.e., out of core) using

Numpy semantics. Furthermore, dask includes extensions for Pandas dataframes

(see Sect. 1.7). Roughly speaking, this means that dask understands how to unpack

Python expressions and translate them for a variety of distributed backend data

services upon which the computing takes place. This means that dask separates

the expression of the computation from the particular implementation on a given

backend.

1.3 Matplotlib

Matplotlib is the primary visualization tool for scientific graphics in Python. Like all

great open-source projects, it originated to satisfy a personal need. At the time of its

inception, John Hunter primarily used MATLAB for scientific visualization, but as

he began to integrate data from disparate sources using Python, he realized he needed a Python solution for visualization, so he single-handedly wrote Matplotlib. Since

those early years, Matplotlib has displaced the other competing methods for twodimensional

scientific visualization and today is a very actively maintained project,

even without John Hunter, who sadly passed away in 2012.

John had a few basic requirements for Matplotlib:

• Plots should look publication quality with beautiful text.

• Plots should output Postscript for inclusion within LATEX documents and publication

quality printing.

• Plots should be embeddable in a graphical user interface (GUI) for application

development.

• The code should be mostly Python to allow for users to become developers.

• Plots should be easy to make with just a few lines of code for simple graphs.

Each of these requirements has been completely satisfied and Matplotlib’s capabilities

have grown far beyond these requirements. In the beginning, to ease the transition

fromMATLAB to Python, many of theMatplotlib functions were closely named after

the corresponding MATLAB commands. The community has moved away from

this style and, even though you may still find the old MATLAB-esque style used in

the online Matplotlib documentation.

The following shows the quickest way to draw a plot using Matplotlib and the

plain Python interpreter. Later, we’ll see howto do this even faster using IPython. The

first line imports the requisite module as plt which is the recommended convention.

The next line plots a sequence of numbers generated using Python’s range object.

Note the output list contains a Line2D object. This is an artist in Matplotlib parlance.

Finally, the plt.show() function draws the plot in a GUI figure window.

import matplotlib.pyplot as plt

plt.plot(range(10))

plt.show() # unnecessary in IPython (discussed later)

If you try this in your own plain Python interpreter (and you should!), you will

see that you cannot type in anything further in the interpreter until the figure window

(i.e., something like Fig. 1.1) is closed. This is because the plt.show() function

preoccupies the interpreter with the controls in the GUI and blocks further interaction.

As we discuss below, IPython provides ways to get around this blocking so you can

simultaneously interact with the interpreter and the figure window.3

As shown in Fig. 1.1, the plot function returns a list containing the Line2D object.

More complicated plots yield larger lists filled with artists. The terminology is

that artists draw on the canvas contained in theMatplotlib figure. The final line is the

plt.show function that provokes the embedded artists to render on the Matplotlib

canvas. The reason this is a separate function is that plots may have dozens of complicated

artists and rendering may be a time-consuming task to only be undertaken at the end, when all the artists have been mustered. Matplotlib supports plotting images,

contours, and many others that we cover in detail in the following chapters.

Even though this is the quickest way to draw a plot in Matplotlib, it is not recommended

because there are no handles to the intermediate products of the plot such

as the plot’s axis. While this is okay for a simple plot like this, later on we will see

how to construct complicated plots using the recommended method.

One of the bestways to get startedwith Matplotlib is to browse the extensive online

gallery of plots on the main Matplotlib site. Each plot comes with corresponding

source code that you can use as a starting point for your own plots. In Sect. 1.4, we

discuss special magic commands that make this particularly easy. The annual John

Hunter: Excellence in Plotting Contest provides fantastic, compelling examples of

scientific visualizations that are possible using Matplotlib.

1.3.1 Alternatives to Matplotlib

Even though Matplotlib is the most complete option for script-based plotting, there

are some alternatives for specialized scientific graphics that may be of interest.

If you require real-time data display and tools for volumetric data rendering and

complicated 3D meshes with isosurfaces, then PyQtGraph is an option. PyQtGraph

is a pure-Python graphics and GUI library that depends on Python bindings for the

Qt GUI library (i.e., PySide or PyQt4) and Numpy. This means that the PyQtGraph

relies on these other libraries (especially Qt’s GraphicsView framework) for the

heavy-duty number crunching and rendering. This package is actively maintained,

with solid documentation. You also need to grasp a few Qt-GUI development concepts

to use this effectively.

An alternative that comes from the R community is ggplot which is a Python

port of the ggplot2 package that is fundamental to statistical graphics in R. From

the Python standpoint, the main advantage of ggplot is the tight integration with

the Pandas dataframe, which makes it easy to draw beautifully formatted statistical

graphs. The downside of this package is that it applies un-Pythonic semantics based

on the Grammar of Graphics [2], which is nonetheless a well-thought-out method

for articulating complicated graphs. Of course, because there are two-way bridges

between Python and R via the R2Py module (among others), it is workable to send

Numpy arrays to R for native ggplot2 rendering and then retrieve the so-computed

graphic back into Python. This is a workflow that is lubricated by the Jupyter Notebook

(see below) via the rmagic extension. Thus, it is quite possible to get the best

of both worlds via the Jupyter Notebook and this kind of multi-language workflow

is quite common in data analysis communities.

1.3.2 Extensions to Matplotlib

Initially, to encourage adoption ofMatplotlib fromMATLAB, many of the graphical

sensibilities were adopted from MATLAB to preserve the look and feel for transitioning

users. Modern sensibilities and prettier default plots are possible because

Matplotlib provides the ability to drill down and tweak every element on the canvas.

However, this can be tedious to do and several alternatives offer relief. For statistical

plots, the first place to look is the seaborn module that includes a vast array of

beautifully formatted plots including violin plots, kernel density plots, and bivariate

histograms. The seaborn gallery includes samples of available plots and the corresponding

code that generates them. Note that importing seaborn hijacks the default

settings for all plots, so you have to coordinate this if you only want to use seaborn

for some (not all) of your visualizations in a given session. Note that you can find

the defaults for Matplotlib in the matplotlib.rcParams dictionary.

1.4 IPython

IPython [3] originated as a way to enhance Python’s basic interpreter for smooth

interactive scientific development. In the early days, the most important enhancement was tab completion for dynamic introspection of workspace variables. For example,

you can start IPython at the commandline by typing ipython and then you should

see something like the following in your terminal:

Python 2.7.11 |Continuum Analytics, Inc.| (default, Dec 7 2015, 14:00

Type "copyright", "credits" or "license" for more information.

IPython 4.0.0 -- An enhanced Interactive Python.

? -> Introduction and overview of IPython's features.

%%quickref -> Quick reference.

help -> Python's own help system.

object? -> Details about 'object', use 'object??' for extra details.

In [1]:

Next, creating a string as shown and hitting the TAB key after the dot character

initiates the introspection, showing all the functions and attributes of the string

object in x.

In [1]: x = 'this is a string'

In [2]: x.<TAB>

x.capitalize x.format x.isupper x.rindex x.strip

x.center x.index x.join x.rjust x.swapcase

x.count x.isalnum x.ljust x.rpartition x.title

x.decode x.isalpha x.lower x.rsplit x.translate

x.encode x.isdigit x.lstrip x.rstrip x.upper

x.endswith x.islower x.partition x.split x.zfill

x.expandtabs x.isspace x.replace x.splitlines

x.find x.istitle x.rfind x.startswith

To get help about any of these, you simply add the ? character at the end as shown

below

In [2]: x.center?

Type: builtin\_function\_or\_method

String Form:<built-in method center of str object at 0x03193390>

Docstring:

S.center(width[, fillchar]) -> string

Return S centered in a string of length width. Padding is

done using the specified fill character (default is a space)

and IPython provides the built-in help documentation. Note that you can also get this

documentation with help(x.center) which works in the plain Python interpreter

as well.

The combination of dynamic tab-based introspection and quick interactive help

accelerates development because you can keep your eyes and fingers in one place as

you work. This was the original IPython experience, but IPython has since grown

into a complete framework for delivering a rich scientific computing workflow that

retains and enhances these fundamental features.

1.5 Jupyter Notebook

As you may have noticed investigating Python on the web, most Python users are

web developers, not scientific programmers, meaning that the Python stack is very

well developed for web technologies. The genius of the IPython development team

was to leverage these technologies for scientific computing by embedding IPython

in modern web browsers. In fact, this strategy has been so successful that IPython

has moved into other languages beyond Python such as Julia and R as the Jupyter

project. You can start the Jupyter Notebook with the following commandline:

Terminal> jupyter notebook

After starting the notebook, you should see something like the following in the

terminal:

[I 16:08:21.213 NotebookApp] Serving notebooks from local directory: /home/user

[I 16:08:21.214 NotebookApp] The Jupyter Notebook is running at:

[I 16:08:21.214 NotebookApp] http://localhost:8888/?token=80281f0c324924d34a4e

[I 16:08:21.214 NotebookApp] Use Control-C to stop this server and shut down

The first line reveals where Jupyter looks for default settings. The next line shows

where it looks for documents in the Jupyter Notebook format. The third line shows

that the Jupyter Notebook started aweb server on the local machine (i.e., 127.0.0.1)

on port number 8888. This is the address your browser needs to connect to the

Jupyter session although your default browser should have opened automatically to

this address. The port number and other configuration options are available either

on the commandline or in the profile file shown in the first line. If you are on a

Windows platform and you do not get this far, then the Window’s firewall is probably

blocking the port. For additional configuration help, see the main Jupyter site

(www.jupyter.org).

When Jupyter starts, it initiates several Python processes that use the blazingfast

ZeroMQ message passing framework for interprocess communication, along

with the web-sockets protocol for back-and-forth communication with the browser.

To start Jupyter and get around your default browser, you can use the additonal

--no-browser flag and then manually type in the local host address

http://127.0.0.1:8888 into your favorite browser to get started. Once all that is

settled, you should see something like the following Fig. 1.2,

You can create a new document by clicking the New Notebook button shown

in Fig. 1.2. Then, you should see something like Fig. 1.3. To start using the Jupyter

Notebook, you just start typing code in the shaded textbox and then hit SHIFT+ENTER

to execute the code in that Jupyter cell. Figure 1.4 shows the dynamic introspection

in the pulldown menu when you type the TAB key after the x.. Context-based help is

also available as before by using the ? suffix which opens a help panel at the bottom

of the browser window. There are many amazing features including the ability to

share notebooks between different users and to run Jupyter Notebooks in the Amazon

cloud, but these features go beyond our scope here. Check the jupyter.org website

or peek at the mailing list for the latest work on these fronts.

The Jupyter Notebook supports high-quality mathematical typesetting using

MathJaX, which is a JavaScript implementation of most of LATEX, as well as video

and other rich content. The concept of consolidating mathematical algorithm descriptions

and the code that implements those algorithms into a shareable document

is more important than all of these amazing features. There is no understating the

importance of this in practice because the algorithm documentation (if it exists) is

usually in one format and completely separate from the code that implements it.

This common practice leads to un-synchronized documentation and code that renders

one or the other useless. The Jupyter Notebook solves this problem by putting

everything into a living shareable document based upon open standards and freely available software. Jupyter Notebooks can even be saved as static HTML documents

for those without Python!

Finally, Jupyter provides a large set of magic commands for creating macros,

profiling, debugging, and viewing codes. A full list of these can be found by typing

in %lsmagic in Jupyter. Help on any of these is available using the ? character suffix.

Some frequently used commands include the %cd command that changes the current

working directory, the %ls command that lists the files in the current directory, and the

%hist command that shows the history of previous commands (including optional

searching). The most important of these for new users is probably the %loadpy

command that can load scripts from the local disk or from the web. Using this to

explore the Matplotlib gallery is a great way to experiment with and re-use the plots

there.

1.6 Scipy

Scipy was the first consolidated module for a wide range of compiled libraries, all

based on Numpy arrays. Scipy includes numerous special functions (e.g., Airy,

Bessel, elliptical) as well as powerful numerical quadrature routines via the QUADPACK

Fortran library (see scipy.integrate), where you will also find other

quadrature methods. Note that some of the same functions appear in multiple places within Scipy itself as well as in Numpy. Additionally, Scipy provides access to the

ODEPACK library for solving differential equations. Lots of statistical functions,

including random number generators, and a wide variety of probability distributions

are included in the scipy.stats module. Interfaces to the Fortran MINPACK optimization

library are provided via scipy.optimize. These include methods for

root-finding, minimization and maximization problems, with and without higher order

derivatives. Methods for interpolation are provided in the scipy.interpolate

module via the FITPACK Fortran package. Note that some of the modules are so

big that you do not get all of them with import scipy because that would take too

long to load. You may have to load some of these packages individually as import

scipy.interpolate, for example.

As we discussed, the Scipy module is already packed with an extensive list of

scientific codes. For that reason, the scikits modules were originally established

as a way to stage candidates that could eventually make it into the already stuffed

Scipy module, but it turns out that many of these modules became so successful on

their own that they will never be integrated into Scipy proper. Some examples include

sklearn for machine learning and scikit-image for image processing.

1.7 Pandas

Pandas [4] is a powerful module that is optimized on top of Numpy and provides

a set of data structures particularly suited to time series and spreadsheet-style data

analysis (think of pivot tables in Excel). If you are familiar with the R statistical

package, then you can think of Pandas as providing a Numpy-powered dataframe for

Python.

1.7.1 Series

There are two primary data structures in Pandas. The first is the Series object which

combines an index and corresponding data values.

>>> import pandas as pd # recommended convention

>>> x=pd.Series(index = range(5),data=[1,3,9,11,12])

>>> x

0 1

1 3

2 9

3 11

4 12

dtype: int64

The main thing to keep in mind with Pandas is that these data structures were originally

designed to work with time-series data. In that case, the index in the data

structures corresponds to a sequence of ordered time stamps. In the general case, the

index must be a sort-able array-like entity. For example,

>>> x=pd.Series(index = ['a','b','d','z','z'],data=[1,3,9,11,12])

>>> x

a 1

b 3

d 9

z 11

z 12

dtype: int64

Note the duplicated z entries in the index. We can get at the entries in the Series

in a number of ways. First, we can used the dot notation to select as in the following:

>>> x.a

1

>>> x.z

z 11

z 12

dtype: int64

We can also use the indexed position of the entries with iloc as in the following:

>>> x.iloc[:3]

a 1

b 3

d 9

dtype: int64

which uses the same slicing syntax as Numpy arrays. You can also slice across the

index, even if it is not numeric with loc as in the following:

>>> x.loc['a':'d']

a 1

b 3

d 9

dtype: int64

which you can get directly from the usual slicing notation:

>>> x['a':'d']

a 1

b 3

d 9

dtype: int64

Note that, unlike Python, slicing this way includes the endpoints. While that is

very interesting, the main power of Pandas comes from its power to aggregate and

group data. In the following, we build a more interesting Series object:

>>> x = pd.Series(range(5),[1,2,11,9,10])

and then group it in the following:

>>> grp=x.groupby(lambda i:i%2) # odd or even

>>> grp.get\_group(0) # even group

2 1

10 4

dtype: int64

>>> grp.get\_group(1) # odd group

1 0

11 2

9 3

dtype: int64

The first line groups the elements of the Series object by whether or not the index

is even or odd. The lambda function returns 0 or 1 depending on whether or not the

corresponding index is even or odd, respectively. The next line shows the 0 (even)

group and then the one after shows the 1 (odd) group. Now, that we have separate

groups,we can perform awide variety of summarizations on the group.You can think

of these as reducing each group into a single value. For example, in the following,

we get the maximum value of each group:

>>> grp.max() # max in each group

0 4

1 3

dtype: int64

Note that the operation above returns another Series object with an index corresponding

to the [0,1] elements.

1.7.2 Dataframe

The Pandas DataFrame is an encapsulation of the Series that extends to two dimensions.

One way to create a DataFrame is with dictionaries as in the following:

>>> df = pd.DataFrame({'col1': [1,3,11,2], 'col2': [9,23,0,2]})

Note that the keys in the input dictionary are now the column headings (labels) of

the DataFrame, with each corresponding column matching the list of corresponding

values from the dictionary. Like the Series object, the DataFrame also has in

index, which is the [0,1,2,3] column on the far-left. We can extract elements

from each column using the iloc method as discussed earlier as shown below

>>> df.iloc[:2,:2] # get section

col1 col2

0 1 9

1 3 23

or by directly slicing or by using the dot notation as shown below

>>> df['col1'] # indexing

0 1

1 3

2 11

3 2

Name: col1, dtype: int64

>>> df.col1 # use dot notation

0 1

1 3

2 11

3 2

Name: col1, dtype: int64

Subsequent operations on the DataFrame preserve its column-wise structure as in

the following:

>>> df.sum()

col1 17

col2 34

dtype: int64

where each column was totaled. Grouping and aggregating with the dataframe is

even more powerful than with Series. Let’s construct the following dataframe:

>>> df = pd.DataFrame({'col1': [1,1,0,0], 'col2': [1,2,3,4]})

In the above dataframe, note that the col1 column has only two entries. We can

group the data using this column as in the following:

>>> grp=df.groupby('col1')

>>> grp.get\_group(0)

col1 col2

2 0 3

3 0 4

>>> grp.get\_group(1)

col1 col2

0 1 1

1 1 2

Note that each group corresponds to entries for which col1 was either of its two

values. Now that we have grouped on col1, as with the Series object, we can also

functionally summarize each of the groups as in the following:

>>> grp.sum()

col2

col1

0 7

1 3

where the sum is applied across each of the Dataframes present in each group. Note

that the index of the output above is each of the values in the original col1.

The Dataframe can compute new columns based on existing columns using the

eval method as shown below

>>> df['sum\_col']=df.eval('col1+col2')

>>> df

col1 col2 sum\_col

0 1 1 2

1 1 2 3

2 0 3 3

3 0 4 4

Note that you can assign the output to a new column to the Dataframe as shown.4

We can group by multiple columns as shown below

>>> grp=df.groupby(['sum\_col','col1'])

Doing the sum operation on each group gives the following:

>>> res=grp.sum()

>>> res

col2

sum\_col col1

2 1 1

3 0 3

1 2

4 0 4

This output is much more complicated than anything we have seen so far, so let’s

carefully walk through it. Below the headers, the first row 2 1 1 indicates that for

sum\_col=2 and for all values of col1 (namely, just the value 1), the value of col2

is 1. For the next row, the same pattern applies except that for sum\_col=3, there are

now two values for col1, namely 0 and 1, which each have their corresponding two

values for the sum operation in col2. This layered display is one way to look at the

result. Note that the layers above are not uniform. Alternatively, we can unstack

this result to obtain the following tabular view of the previous result:

>>> res.unstack()

col2

col1 0 1

sum\_col

2 NaN 1.0

3 3.0 2.0

4 4.0 NaN

The NaN values indicate positions in the tablewhere there is no entry. For example, for

the pair (sum\_col=2,col2=0), there is no corresponding value in the Dataframe,

as you may verify by looking at the penultimate code block. There is also no entry

corresponding to the (sum\_col=4,col2=1) pair. Thus, this shows that the original

presentation in the penultimate code block is the same as this one, just without the

abovementioned missing entries indicated by NaN.

We have barely scratched the surface of what Pandas is capable of and we have

completely ignored its powerful features for managing dates and times. The text by

Mckinney [4] is a very complete and happily readable introduction to Pandas. The

online documentation and tutorials at the main Pandas site are also great for diving

deeper into Pandas.

1.8 Sympy

Sympy [5] is the main computer algebra module in Python. It is a pure-Python

package with no platform dependencies. With the help of multiple Google Summer

of Code sponsorships, it has grown into a powerful computer algebra system with

many collateral projects that make it faster and integrate it tighter with Numpy and

Jupyter. Sympy’s online tutorial is excellent and allows interactingwith its embedded

code samples in the browser by running the code on the Google App Engine behind

the scenes. This provides an excellent way to interact and experiment with Sympy.

If you find Sympy too slow or need algorithms that it does not implement, then

SAGE is your next stop. The SAGE project is a consolidation of over 70 of the

best open-source packages for computer algebra and related computation. Although

Sympy and SAGE share code freely between them, SAGE is a specialized build of

the Python kernel to facilitate deep integration with the underlying libraries. Thus,

it is not a pure-Python solution for computer algebra (i.e., not as portable) and it is a

proper superset of Python with its own extended syntax. The choice between SAGE

and Sympy really depends on whether or not you intend primarily work in SAGE or

just need occasional computer algebra support in your existing Python code.

An important new development regarding SAGE is the freely available SAGE

Cloud (https://cloud.sagemath.com/), sponsored by University of Washington that

allows you to use SAGE entirely in the browser with no additional setup. Both

SAGE and Sympy offer tight integrationwith the Jupyter Notebook formathematical

typesetting in the browser using MathJaX.

To get started with Sympy, you must import the module as usual,

>>> import sympy as S # might take awhile

which may take a bit because it is a big package. The next step is to create a Sympy

variable as in the following:

>>> x = S.symbols('x')

Now we can manipulate this using Sympy functions and Python logic as shown

below

>>> p=sum(x\*\*i for i in range(3)) # 2nd order polynomial

>>> p

x\*\*2 + x + 1

Now, we can find the roots of this polynomial using Sympy functions,

>>> S.solve(p) # solves p == 0

[-1/2 - sqrt(3)\*I/2, -1/2 + sqrt(3)\*I/2]

There is also a sympy.roots function that provides the same output but as a dictionary.

>>> S.roots(p)

{-1/2 - sqrt(3)\*I/2: 1, -1/2 + sqrt(3)\*I/2: 1}

We can also have more than one symbolic element in any expression as in the following:

>>> from sympy.abc import a,b,c # quick way to get common symbols

>>> p = a\* x\*\*2 + b\*x + c

>>> S.solve(p,x) # specific solving for x-variable

[(-b + sqrt(-4\*a\*c + b\*\*2))/(2\*a), -(b + sqrt(-4\*a\*c + b\*\*2))/(2\*a)]

which is the usual quadratic formula for roots. Sympy also provides many mathematical

functions designed to work with Sympy variables. For example,

>>> S.exp(S.I\*a) #using Sympy exponential

exp(I\*a)

We can expand this using expand\_complex to obtain the following:

>>> S.expand\_complex(S.exp(S.I\*a))

I\*exp(-im(a))\*sin(re(a)) + exp(-im(a))\*cos(re(a))

which gives us Euler’s formula for the complex exponential. Note that Sympy does

not know whether or not a is itself a complex number. We can fix this by making

that fact part of the construction of a as in the following:

>>> a = S.symbols('a',real=True)

>>> S.expand\_complex(S.exp(S.I\*a))

I\*sin(a) + cos(a)

Note the much simpler output this time because we have forced the additional condition

on a.

A powerful way to use Sympy is to construct complicated expressions that you

can later evaluate using Numpy via the lambdify method. For example,

>>> y = S.tan(x) \* x + x\*\*2

>>> yf= S.lambdify(x,y,'numpy')

>>> y.subs(x,.1) # evaluated using Sympy

0.0200334672085451

>>> yf(.1) # evaluated using Numpy

0.020033467208545055

After creating the Numpy function with lambdify, you can use Numpy arrays as

input as shown

>>> yf(np.arange(3)) # input is Numpy array

array([ 0. , 2.55740772, -0.37007973])

>>> [ y.subs(x,i).evalf() for i in range(3) ] # need extra work for Sympy

[0, 2.55740772465490, -0.370079726523038]

We can get the same output using Sympy, but that requires the extra programming

logic shown to do the vectorizing that Numpy performs natively.

Once again, we have merely scratched the surface of what Sympy is capable of

and the online interactive tutorial is the best place to learn more. Sympy also allows

automatic mathematical typesetting within the Jupyter Notebook using LATEX so the

so-constructed notebooks look almost publication-ready (see sympy.latex) and

can be made so with the jupyter nbconvert command. This makes it easier to

jump the cognitive gap between the Python code and the symbology of traditional

mathematics.

1.9 Interfacing with Compiled Libraries

As we have discussed, Python for scientific computing really consists of gluing

together different scientific libraries written in a compiled language like C or Fortran.

Ultimately, you maywant to use libraries not availablewith existing Python bindings.

There are many, many options for doing this. The most direct way is to use the builtin

ctypes module which provides tools for providing input/output pointers to the

library’s functions just as if you were calling them from a compiled language. This

means that you have to know the function signatures in the library exactly—how

many bytes for each input and how many bytes for the output. You are responsible

for building the inputs exactly the way the library expects and collecting the resulting outputs. Even though this seems tedious, Python bindings for vast libraries have been

built this way.

If you want an easier way, then SWIG is an automatic wrapper generating tool

that can provide bindings to a long list of languages, not just Python; so if you need

bindings for multiple languages, then this is your best and only option. Using SWIG

consists of writing an interface file so that the compiled Python dynamically linked

library (.pyd files) can be readily imported into the Python interpreter. Huge and

complex libraries like Trilinos (SandiaNational Labs) have been interfaced to Python

using SWIG, so it is a well-tested option. SWIG also supports Numpy arrays.

However, the SWIG model assumes that youwant to continue developing primarily

in C/Fortran and you are hooking into Python for usability or other reasons. On the

other hand, if you start developing algorithms in Python and then want to speed them

up, then Cython is an excellent option because it provides a mixed language that

allows you to have both C language and Python code intermixed. Like SWIG, you

have to write additional files in this hybrid Python/C dialect to have Cython generate

the C-code that you will ultimately compile. The best part of Cython is the profiler

that can generate an HTML report showing where the code is slow and could benefit

from translation to Cython. The Jupyter Notebook integrates nicely with Cython

via its %cython magic command. This means you can write Cython code in a cell in

Jupyter Notebook and the notebook will handle all of the tedious details like setting

up the intermediate files to actually compile the Cython extension. Cython also

supports Numpy arrays.

Cython and SWIG are just two of the ways to create Python bindings for your

favorite compiled libraries. Other notable (but less popular) options include FWrap,

f2py, CFFI, and weave. It is also possible to use Python’s own API directly, but

this is a tedious undertaking that is hard to justify given the existence of so many

well-developed alternatives.

1.10 Integrated Development Environments

For those who prefer integrated development environments (IDEs), there is a lot to

choose from. The most comprehensive is Enthought Canopy, which includes a rich,

syntax-highlighted editor, integrated help, debugger, and even integrated training. If

you are already familiar with Eclipse from other projects, or do mixed-language programming,

then there is a Python plug-in called PyDev that contains all usual features

from Eclipse with a Python debugger.Wingware provides an affordable professionallevel

IDE with multi-project management support and unusually clairvoyant code

completion that works even in debug mode. Another favorite is PyCharm, which also

supports multiple languages and is particularly popular among Python web developers

because it provides powerful templates for popular web frameworks like Django.

Visual Studio Code has quickly developed a strong following among Python newcomers

because of its beautiful interface and plug-in ecosystem. If you are a VIM

user, then the Jedi plug-in provides excellent code completion that works well with pylint, which provides static code analysis (i.e., identifies missing modules and

typos). Naturally, emacs has many related plug-ins for developing in Python. Note

that are many other options, but I have tried to emphasize those most suitable for

Python beginners.

1.11 Quick Guide to Performance and Parallel

Programming

There are many options available to improve the performance of your Python codes.

The first thing to determine is what is limiting your computation. It could be CPU

speed (unlikely), memory limitations (out-of-core computing), or it could be data

transfer speed (waiting on data to arrive for processing). If your code is pure-Python,

then you can try running it with Pypy, which is is an alternative Python implementation

that employs a just-in-time compiler. If your code does not experience a massive

speedup with Pypy, then there is probably something external to the code that is

slowing it down (e.g., disk access or network access). If Pypy doesn’t make any

sense because you are using many compiled modules that Pypy does not support,

then there are many diagnostic tools available.

Python has its own built-in profiler cProfile you can invoke from the command

line as in the following:

>>> python -m cProfile -o program.prof my\_program.py

The output of the profiler is saved to the program.prof file. This file can be visualized

in runsnakerun to get a nice graphical picture of where the code is spending

the most time. The task manager on your operating system can also provide clues as

your program runs to see how it is consuming resources. The line\_profiler by

Robert Kern provides an excellent way to see how the code is spending its time by

annotating each line of the code by its timings. In combination with runsnakerun,

this narrows down problems to the line level from the function level.

The most common situation is that your program is waiting on data from disk

or from some busy network resource. This is a common situation in web programming

and there are lots of well-established tools to deal with this. Python has a

multiprocessing module that is part of the standard library. This makes it easy

to spawn child worker processes that can break off and individually process small

parts of a big job. However, it is still your responsibility as the programmer to figure

out how to distribute the data for your algorithm. Using this module means that the

individual processes are to be managed by the operating system, which will be in

charge of balancing the load.

The basic template for using multiprocessing is the following:

# filename multiprocessing\_demo.py

import multiprocessing

import time

def worker(k):

'worker function'

print('am starting process %d' % (k))

time.sleep(10) # wait ten seconds

print('am done waiting!')

return

if \_\_name\_\_ == '\_\_main\_\_':

for i in range(10):

p = multiprocessing.Process(target=worker, args=(i,))

p.start()

Then, you run this program at the terminal as in the following:

Terminal> python multiprocessing\_demo.py

It is crucially important that you run the program from the terminal this way. It is not

possible to do this interactively from within Jupyter, say. If you look at the process

manager on the operating system, you should see a number of new Python processes

loitering for ten seconds. You should also see the output of the print statements

above. Naturally, in a real application, you would be assigning some meaningful

work for each of the workers and figuring out how to send partially finished pieces

between individual workers. Doing this is complex and easy to get wrong, so Python

3 has the helpful concurrent.futures.

# filename: concurrent\_demo.py

from concurrent import futures

import time

def worker(k):

'worker function'

print ('am starting process %d' % (k))

time.sleep(10) # wait ten seconds

print ('am done waiting!')

return

def main():

with futures.ProcessPoolExecutor(max\_workers=3) as executor:

list(executor.map(worker,range(10)))

if \_\_name\_\_ == '\_\_main\_\_':

main()

Terminal> python concurrent\_demo.py

You should see something like the following in the terminal. Note that we explicitly

restricted the number of processes to three.

am starting process 0

am starting process 1

am starting process 2

am done waiting!

am done waiting!

...

The futures module is built on top of multiprocessing and makes it easier

to use for this kind of simple task. Note that there are also versions of both that use

threads instead of processes while maintaining the same usage pattern. The main difference between threads and processes is that processes have their own compartmentalized

resources. The C language Python (i.e., CPython) implementation uses

a global interpreter lock (GIL) that prevents threads from locking up on internal

data structures. This is a course-grained locking mechanism where one thread may

individually run faster because it does not have to keep track of all the bookkeeping

involved in running multiple threads simultaneously. The downside is that you

cannot run multiple threads simultaneously to speed up certain tasks.

There is no corresponding locking problem with processes but these are somewhat

slower to start up because each process has to create its own private workspace

for data structures that may be transferred between them. However, each process

can certainly run independently and simultaneously once all that is set up. Note

that certain alternative implementations of Python like IronPython use a finer-grain

threading design rather than aGIL approach. As a final comment, onmodern systems

with multiple cores, it could be that multiple threads actually slow things down

because the operating system may have to switch threads between different cores.

This creates additional overheads in the thread switching mechanism that ultimately

slow things down.

Jupyter itself has a parallel programming framework built (ipyparallel) that is

both powerful and easy to use. The first step is to fire up separate Jupyter engines at

the terminal as in the following:

Terminal> ipcluster start --n=4

Then, in an Jupyter window, you can get the client,

In [1]: from ipyparallel import Client

...: rc = Client()

The client has a connection to each of the processes we started before using

ipcluster. To use all of the engines, we assign the DirectView object from the

client as in the following:

In [2]: dview = rc[:]

Now, we can apply functions for each of the engines. For example, we can get the

process identifiers using the os.getpid function,

In [3]: import os

In [4]: dview.apply\_sync(os.getpid)

Out[4]: [6824, 4752, 8836, 3124]

Once the engines are up and running, data can be distributed to them using scatter,

In [5]: dview.scatter('a',range(10))

Out[5]: <AsyncResult: finished>

In [6]: dview.execute('print(a)').display\_outputs()

[stdout:0] [0, 1, 2]

[stdout:1] [3, 4, 5]

[stdout:2] [6, 7]

[stdout:3] [8, 9]

Note that the execute method evaluates the given string in each engine. Now that

the data have been sprinkled among the active engines, we can do further computing

on them,

In [7]: dview.execute('b=sum(a)')

Out[7]: <AsyncResult: finished>

In [8]: dview.execute('print(b)').display\_outputs()

[stdout:0] 3

[stdout:1] 12

[stdout:2] 13

[stdout:3] 17

In this example, we added up the individual a sub-lists available on each of the

engines.We can gather up the individual results into a single list as in the following:

In [9]: dview.gather('b').result

Out[9]: [3, 12, 13, 17]

This is one of the simplest mechanisms for distributing work to the individual engines

and collecting the results. Unlike the other methods we discussed, you can do

this iteratively, which makes it easy to experiment with how you want to distribute

and compute with the data. The Jupyter documentation has many more examples

of parallel programming styles that include running the engines on cloud resources,

supercomputer clusters, and across disparate networked computing resources. Although

there are many other specialized parallel programming packages, Jupyter

provides the best trade-off for generality against complexity across all of the major

platforms.

1.12 Other Resources

The Python community is filled with super-smart and amazingly helpful people. One

of the best places to get help with scientific Python is the www.stackoverflow.com

site which hosts a competitive Q&A forum that is particularly welcoming for Python

newbies. Several of the key Python developers regularly participate there and the

quality of the answers is very high. The mailing lists for any of the key tools (e.g.,

Numpy, Jupyter, Matplotlib) are also great for keeping up with the newest developments.

Anything written by Hans Petter Langtangen [6] is excellent, especially if

you have a physics background. The Scientific Python conference held annually in

Austin is also a great place to see your favorite developers in person, ask questions,

and participate in the many interesting subgroups organized around niche topics.

The PyData workshop is a semi-annual meeting focused on Python for large-scale

data-intensive processing.

References

1. T.E. Oliphant, A Guide to NumPy (Trelgol Publishing, 2006)
2. L. Wilkinson, D. Wills, D. Rope, A. Norton, R. Dubbs, The Grammar of Graphics. Statistics and Computing (Springer, Berlin, 2006)
3. F. Perez, B.E. Granger et al., IPython software package for interactive scientific computing. <http://ipython.org/>
4. W. McKinney, Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython (O’Reilly, 2012)
5. O. Certik et al., SymPy: python library for symbolic mathematics. <http://sympy.org/>
6. H.P. Langtangen, Python Scripting for Computational Science, vol. 3, 3rd edn. Texts in Computational Science and Engineering (Springer, Berlin, 2009)