Python como entorno de desarrollo científico.

Guillem Borrell Nogueras

8 de julio de 2008

1. Introducción

Existe cierta confusión con los conceptos de lenguaje y herramienta. Mientras una herramienta nace como respuesta a una necesidad funcional los lenguajes sirven para dar sentido a la realidad; se encuentran en planos completamente distintos. Los lenguajes de programación tienen más en común con las lenguas o el lenguaje matemático que con un mecanismo o un motor. Sin embargo uno puede ahogarse en libros donde se analiza sistemáticamente un lenguaje de programación como si de una herramienta se tratara.

Los parámetros para juzgar un lenguaje son ajenos a un ingeniero o un matemático y más cercanos a un filósofo: aprendizaje, incertidumbre, consistencia o formalismo. La programación es la creación de literatura con un lenguaje específico y tiene ingredientes como la sintaxis, el léxico... ¡Incluso faltas de ortografía! Un ingeniero tiende a valorar según parámetros como la eficiencia, la simplicidad o la rapidez. Pocas veces constatamos el error que estos conceptos no pueden aplicarse a un lenguaje porque son propios de los mecanismos. Es algo bastante común: cuando sólo tienes un martillo todo te parece un clavo.

Los lenguajes de programación evolucionan como las lenguas naturales. Nacen a partir de lenguas primitivas y siguen uno de los dos caminos posibles: la evolución o la muerte. Existen en ambos casos lenguas muertas. Podría compararse COBOL con el latín, a pocos se les ocurriría aprenderlos hoy en día sin embargo los programas que controlan las transacciones bancarias se escriben en COBOL al igual que en el Vaticano se siguen analizando textos en latín.

Los lenguajes sirven para comunicar significado. Utilizar el castellano para una crónica futbolística es equivalente a formular un teorema mediante un lenguaje matemático. Los lenguajes de programación sirven para comunicar algoritmos a ordenadores. Aquí aparece otro concepto crucial: comunicación. Es una acción que requiere un interlocutor y en la programación siempre es el mismo: un ordenador.

La comunicación con los ordenadores ha evolucionado rápidamente desde la construcción del primer ordenador programable. Una anécdota posiblemente apócrifa cuenta que Seymour Cray[1] era capaz de arrancar el sistema operativo de un CDC7600 manipulando la memoria de este ordenador primitivo a mano. Ahora los lenguajes de programación son tan simples que hasta un niño de ocho años es capaz de aprenderlos[2]. ¿Quién se esforzaría hoy en depurar un programa a partir del volcado de memoria? Hablarle a los ordenadores es más sencillo porque se han vuelto más listos, han evolucionado.

Python es fruto de esta evolución. Es un lenguaje de programación de quinta o sexta generación nacido cuando los ordenadores eran ya tan potentes como para no tener que estar continuamente pensando en la memoria y la velocidad de ejecución. Es uno de los primeros lenguajes de programación en los que su propia naturaleza no se ha visto coartada por las limitaciones del ordenador. Es, en consecuencia, uno de los lenguajes más humanos con los que se puede programar. Hoy pocos discuten que una herramienta más cercana al programador ayuda a implementar algoritmos más complejos en menos tiempo y a cometer

[1] `http://en.wikipedia.org/wiki/Seymour_Cray`

[2] Un texto clásico sobre la evolución de los lenguajes de programación es *Real men don't use Pascal*: `http://www.pbm.com/~lindahl/real.programmers.html`

menos errores. Esto es fundamental para que quienes no disponen de una formación específica en la programación, como científicos e ingenieros, escriban programas sin tener que aprender más allá de lo fundamental.

Los lenguajes de programación de alto nivel deben traducirse mediante compilador o un intérprete. Este es el punto donde se crea la confusión: el compilador o el intérprete sí son herramientas desde el punto de vista técnico. La naturaleza de un lenguaje de programación influye significativamente en el diseño y posibilidades del compilador o del intérprete. Por ejemplo, en los lenguajes dinámicos como Python el tipo de las variables se conoce en tiempo de ejecución. Esto añade ciertas posibilidades al lenguaje como el hecho de no tener que declarar las variables. Muchos de estos lenguajes no pueden ser compilados lo que ya impone que la comunicación nunca podrá efectuarse con un compilador sino que tendrá que ser con un intérprete.

Un ingeniero no tiene la formación necesaria para analizar un lenguaje pero sí puede entrar en la discusión sobre qué maquinaria, ya sea interprete o compilador, es más adecuada en cada caso. Es en este punto donde el lenguaje pasa a un segundo plano, por ejemplo: utilizar las posibilidades de optimización de un compilador de C, aparcando por ello Python, a pesar de sus peores características como lenguaje. Es la suma todas las herramientas necesarias para resultar productivo lo que influye en el proceso de decisión.

Este artículo pretende analizar cada una de las ventajas de utilizar Python en un entorno científico y técnico, describir los posibles inconvenientes y proponer soluciones para minimizar su efecto.

Programar en Python es más efectivo porque es más potente sin ser más complejo. Las razones por las que no se ha impuesto aún en un entorno científico y técnico son las siguientes:

- El desconocimiento.
- Que el intérprete no se comporte de la manera adecuada.

Si ya estamos convencidos de la superioridad como lenguaje de Python ¿Es realmente necesario desperdiciar sus bondades como lenguaje por culpa de las características del intérprete? El objetivo de este ensayo es *demostrar que Python dispone en la actualidad de una colección de herramientas suficientes como para minimizar la mayoría de los inconvenientes que podrían descartar su uso*.

1.1. Un enfoque distinto para cada problema

Las aplicaciones de simulación pueden dividirse en dos grandes grupos según sus necesidades computacionales.

- Los pequeños programas de entre una decena y el millar de líneas de código con la misión de realizar un cálculo relativamente simple llamados guiones o *scripts*.
- Las grandes simulaciones de computación de alto rendimiento con tiempos de ejecución largos en superordenadores y ciclos de vida de años o décadas.

Los *scripts* suelen implementarse en lenguajes interpretados especializados en matemáticas e ingeniería, la mayoría de ellos propietarios, que permiten un acceso sencillo y directo a una enorme biblioteca de funciones especializadas[3]. En este ámbito Python compite directamente con Matlab, Mathematica, Maple, IDL... En estos casos Python es ya una alternativa a este software ya que ofrece funcionalidades muy parecidas a coste cero. Esto significa que Python debe disponer de una colección de bibliotecas considerable y comparable con los productos comerciales actuales. Las primeras secciones de este artículo

[3]Los lenguajes de *nicho* o diseñados específicamente para un sector reciben el nombre de *Domain Specific Languages*, a partir de ahora DSL

se dedicarán a listar de un modo poco detallado todas las bibliotecas y aplicaciones que han sido escritas en Python o utilizan el intérprete de Python de utilidad en Matemáticas, Física o Ingeniería.

En el otro extremo, tradicionalmente se ha argumentado que debido a que los programas más exigentes desde un punto de vista computacional deben programarse en el lenguaje que asegure una máxima velocidad en la ejecución prescindiendo de ciertas propiedades deseables. Si bien existen unos pocos casos patológicos en los que la velocidad es la única variable a tener en cuenta[4], en la mayoría la parte del código que explota al máximo hardware se reduce a unas pocas líneas. Este artículo dedicará sus últimas scciones a comentar las distintas técnicas de optimización de código y qué posibilidades ofrece Python para ello.

Atención!

Este documento en pdf contiene archivos adjuntos necesarios para realizar los ejemplos descritos.

2. Scripting en Python

Python fue diseñado como un lenguaje de scripting para uso general. El principal criterio de diseño se resume en una frase: *en Python sólo debe existir una manera elegante y lógica de hacer cualquier cosa*. Esta condición de diseño tan poco precisa se complementa con lo siguiente:

- Ser lo suficientemente simple como para que pueda recordarse fácilmente.
- Soportar todos los paradigmas modernos de programación.
- Ofrecer una librería estándar amplia que responda a la mayoría de las necesidades.
- Forzar una sintaxis clara y un código leíble y fácilmente modificable.

La programación en Python se ha convertido paulatinamente en la expresión del minimalismo en la programación y queda reflejada en el código *zen* de Python. Es una buena oportunidad para ejecutar el primer comando dentro de una consola de python[5].

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
```

[4] Algunos códigos de Mecánica de Fluidos Computacional tienen tiempos de ejecución de meses o incluso años y ciclos de desarrollo de entre dos y cuatro años. Es comprensible que en estos casos no se tengan en cuenta muchas de las técnicas que se proponen en este artículo.

[5] Para ejecutar este comando debe disponerse de un intérprete de Python. Si no está instalado en el sistema puede seguirse el tutorial del apéndice C.

```
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Estas características han convertido Python en el entorno de desarrollo ideal para quienes no tienen conocimientos específicos sobre entornos de desarrollo pero sí saben programar como suele ser habitual entre científicos e ingenieros. Su crecimiento en el ámbito de la enseñanza es lento pero constante y viene confirmado por la progresiva aparición de proyectos científicos que utilizan Python como lenguaje de programación.

Aunque es muy adecuado para iniciarse en la programación aún no se ha popularizado lo suficiente como para que se enseñe en las universidades. Consecuencia directa de ello es que la gran mayoría de quienes se interesan por Python lo hacen ya habiendo aprendido otros lenguajes. Si se posee cierta experiencia programando Python es muy fácil de aprender, la documentación dispone de un tutorial que puede completarse en un par de horas y asimilarse en unos pocos días. Dispone de una consola interactiva completamente funcional y es de gran ayuda para experimentar o manipular directamente resultados.

En UNIX la manera más sencilla de iniciar una consola es mediante el terminal, simplemente escribiendo *python* en él.

```
$ python
Python 2.5.2 (r252:60911, Jun 28 2008, 04:30:43)
[GCC 4.3.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

En Windows el instalador oficial proporciona también una que puede arrancarse a partir de la entrada correspondiente en el menú de inicio.

Cualquier instrucción de código Python es entendida del mismo modo por el intérprete y por la consola. Esta es, por ejemplo, la manera más sencilla de sacar por pantalla la serie de Fibonacci:

```
>>> (a,b)=(0,1)
>>> while b < 10:
...     print b
...     (a,b)=(b,a+b)
...
1
1
2
3
5
8
```

Los programadores con cierta experiencia habrán distinguido las siguientes particularidades:

- El tipo definido entre paréntesis es un tuple y es una generalización de las variables que funciona también en la asignación.
- La definición de los bloques de ejecución, en este caso un bucle lógico, se definen por el sangrado del propio bloque. Se recomienda que este sangrado esté formado por cuatro espacios para homo-

geneizar todo el código existente aunque el intérprete pueda entender también sangrados de dos espacios y tabulaciones. Esta característica crea tantos adeptos como detractores.

- Representar en pantalla es tan sencillo como escribir la variable tras el comando print.

Tan sencillo como este juego matemático es crear una ventana gráfica sobre la que empezar un interfaz de usuario

```
>>> from Tkinter import *
>>> t=Tk()
```

Acto seguido aparece la ventana representada en la figura 2

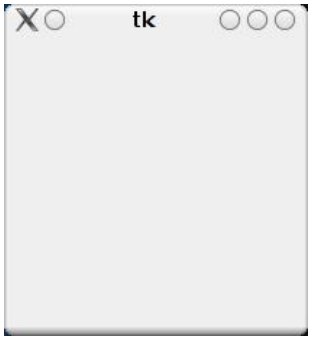

Figura 1: Ventana gracias al módulo Tkinter

También esta vez los programadores expertos habrán notado que el mecanismo para cargar en el intérprete nuevas funciones son los módulos y que a su vez cada módulo es un *namespace*.

La última particularidad digna de mención antes de empezar con las características del lenguaje es mencionar la ayuda interactiva. Cada función de la biblioteca estándar, que analizaremos a continuación, está meticulosamente documentada para que cualquier sesión con el interprete pueda ser además una visita a la documentación. Por ejemplo:

```
>>> import cmath
>>> help(cmath.acos)
Help on built-in function acos in module cmath:

acos(...)
    acos(x)

    Return the arc cosine of x.
```

Puede apreciarse que los módulos no sólo definen un *namespace* sino que además cada módulo es un objeto. A partir de la función *help* puede consultarse la documentación de cada función a través del intérprete.

Recuerda que en los lenguajes interpretados no hay tipos derivados

2.1. El lenguaje de programación Python

Esta es sólo una breve introducción de las sentencias y la sintaxis del lenguaje Python. Es una descripción deliberadamente incompleta de modo que el tutorial oficial del lenguaje[TUT] se convierte a partir de ahora en una lectura obligada.

Para dar un poco más de profundidad a este análisis tan breve todos los tipos y las sentencias se compararán con las del lenguaje de *scripting* por excelencia dentro del entorno científico y técnico, Matlab.

2.1.1. Operaciones aritméticas

Python puede utilizarse como una calculadora algebraica para realizar las operaciones aritméticas usuales:

```
>>> 2+2 #Suma
4
>>> 2-1 #Resta
1
>>> 2*4.5 #Multiplicacion
9.0
>>> 2/4.5 #Division
0.44444444444444442
```

Las únicas diferencias entre Python y Matlab es que hasta la versión 3.x la divisón entre dos enteros devuelve otro entero:

```
>>> 5/3
1
```

Para conseguir en la versión 2.x el mismo comportamiento que en la 3.x bastará con introducir los siguientes comandos en el intérprete o con incluirlos en el código fuente.

```
>>> from __future__ import division
>>> 5/3
1.6666666666666667
```

Y que el operador potencia obedece al doble asterisco como en Fortran mientras que el símbolo de acento circunflejo se reserva para el operador *XOR* como en C.

```
>>> 5**3
125
```

Al igual que en Matlab la representación por omisión de los números en coma flotante es la de doble precisión, es decir, el *double* de C.

Las funciones matemáticas báscias se encuentran en el módulo *math* y el soporte para números complejos en *cmath*.

```
>>> import math as m
>>> import cmath as c
>>> a=3.0+4.0j
>>> a/complex(2,1.3)
(1.968365553602812+0.72056239015817214j)
>>> c.exp(_)
(5.3794960137747223+4.7235385533272529j)
>>> m.exp(abs(_))
1285.5809221999864
```

En esta pequeña sección se muestran varias características de Python.

- La función exponencial, *exp*, se encuentra tanto en el módulo *math* para números en coma flotante como en el *cmath* de modo que si sólo se importa la función puede existir un conflicto de nombres. La solución es importar cada módulo en un namespace distinto.
- Se puede generar un número complejo tanto directamente con la constante reservada *j* como mediante la función *complex*.
- El carácter _ es también una constante reservada que almacena el resultado del comando anterior.

2.2. Control de flujo

La gran particularidad de Python para el control de flujo es esa peculiar manera de diferenciar los bloques de ejecución.

2.2.1. `if`

```
>>> x = int(raw_input("Por favor, introduzca un entero: "))
>>> if x < 0:
...       x = 0
...       print 'Era negativo y lo he cambiado a cero'
... elif x == 0:
...       print 'Cero'
... elif x == 1:
...       print 'Uno'
... else:
...       print 'Mas'
...
```

2.2.2. `for`

Python, como en algunos lenguajes interpretados, las iteraciones no son específicamente un bucle sino que se trata de la secuenciación de un iterador.

```
>>> a = ['gato', 'ventana', 'defenestrar']
>>> for x in a:
...     print x, len(x)
...
gato 4
ventana 7
defenestrar 11
>>> for x in a: # Haced esto y morid
...     if len(x) > 6: a.insert(0, x)
```

Es importante no intentar modificar una lista mientras se está iterando sobre ella, hay que crear una copia temporal gracias al *slicing*.

Disponemos también de las sentencias `break` y `continue` para controlar el flujo de ejecución dentro del bucle.

2.2.3. La función `range`

Esta función devuelve el iterable más sencillo posible, un contador

```
>>> range(4)
[0, 1, 2, 3]
>>> range(3,12,3)
[3, 6, 9]
```

2.2.4. Definición de funciones

Ahora modificamos la función que crea la serie de fibonacci para que devuelva una lista:

```
def fib2(n):
    """Devuelve una lista con la serie de Fibonacci hasta n."""
    result = []
```

```
    (a, b) = (0, 1)
    while b < n:
        result.append(b)
        (a, b) = (b, a+b)
    return result
```

Si una función debe devolver más de una variable debe utilizarse un *tuple*. La palabra clave *return* termina también con la ejecución de la función así que es una estrategia más para el control de flujo de ejecución.

El intérprete sabe que la definición de la función se ha acabado porque, al igual que con las otras estructuras de código descritas hasta ahora, termina su nivel de sangrado propio.

2.2.5. Funciones lambda

Especialmente útiles en el caso que uno necesite devolver una función como argumento de un método. Python también soporta algunas utilidades de la programación funcional como los decoradores.

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

2.3. Palabras clave

Una perfecta demostración que Python es un lenguaje *pequeño* es la cantidad mínima de palabras clave que se reservan.

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

2.4. Tipos

Como todos los lenguajes modernos orientados a objetos como los tipos *estándares* a parte de disponen de una gran funcionalidad. El tipo *lista*, por ejmplo, proporciona métodos para realizar casi cualquier operación que se nos pueda ocurrir con una lista. Python dispone de los siguientes:

- Enteros
- Números en coma flotante
- Números complejos
- Cadenas de texto
- Listas

- Tuples
- Conjuntos
- Diccionarios

Cada uno de ellos viene definido por una clase propia dentro del lenguaje con sus método asignados. A continuación se presenta una lista de todos los métodos disponibles dentro de la clase *lista*:

```
a.__add__            a.__iadd__           a.__rmul__
a.__class__          a.__imul__           a.__setattr__
a.__contains__       a.__init__           a.__setitem__
a.__delattr__        a.__iter__           a.__setslice__
a.__delitem__        a.__le__             a.__str__
a.__delslice__       a.__len__            a.append
a.__doc__            a.__lt__             a.count
a.__eq__             a.__mul__            a.extend
a.__ge__             a.__ne__             a.index
a.__getattribute__   a.__new__            a.insert
a.__getitem__        a.__reduce__         a.pop
a.__getslice__       a.__reduce_ex__      a.remove
a.__gt__             a.__repr__           a.reverse
a.__hash__           a.__reversed__       a.sort
```

La descripción detallada de todos los métodos para cada uno de los tipos puede encontrarse en la referencia oficial del lenguaje [REF] que se distribuye conjuntamente con el tutorial mencionado anteriormente. Mientras seguir con atención el tutorial es casi imprescindible para programar la referencia existe únicamente como herramienta de consulta.

Quizás una de las cosas que más caracteriza las listas en Python es su creación mediante los *list comprehensions*

```
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```

Ejercicio:

Crear una lista que contenga las cadenas de texto `uno`, `dos`, `tres` y `cuatro`. Copiar los elementos que contienen la letra *o* a una lista nueva. Primero hacerlo con un bucle y luego mediante un *list comprehension*(una línea de código).

2.5. La biblioteca estándar

Una de las particularidades de Python como lenguaje es la gran extensión temática de su biblioteca estándar. Lejos de mantenerse ajeno de las bibliotecas y aplicaciones muchos de los módulos de uso común se han *oficializado* hasta formar parte de la distribución oficial.

Está perfectamente documentada en [STB]. Dentro de la misma encontramos módulos para:

- Manipulación textual
- Envío de correos electrónicos
- Proceso de contenido en XML, csv, XDR...
- Criptografía
- Interacción con el sistema operativo
- Compresión de datos
- Almacenamiento de variables
- Enlazado con librerías externas.
- Acceso a las funcionalidades POSIX
- IPC y redes.
- Protocolos de internet (XMLRPC, cookies, cgi, smtp, http, ftp...)
- Multimedia
- GUI
- Internacionalización
- Documentación
- Tests
- Debugging
- Profiling

Es una lista considerable para ser una biblioteca estandar si se compara con libc o la STL. Esta lista no tiene en cuenta en ningún caso todos los módulos no oficiales que pueden encontrarse en el Python Package Index `http://pypi.python.org/pypi`.

Ejercicio:

A partir del módulo os ejecutar la orden `ls` para ver el contenido del directorio actual.

Ejercicio:

A partir del módulo sys conseguir que un programa escrito en Python devuelva una ayuda al pasarle el argumento *-h*

3. Orientación a objetos

La orientación a objetos es un paradigma para la implementación de algoritmos ya no tan nuevo. Es una estructura que puede contener a la vez variables y métodos propios y que dispone de ciertas propiedades como la herencia y el polimorfismo.

Su origen proviene de la necesidad de acercar el lenguaje formal de la programación a la realidad. Los tipos habituales como real, entero o cadena de caracteres describen elementos que habitan de forma natural en el ordenador, ninguno de ellos tiene sentido en la naturaleza. Ni siquiera los caracteres puesto que los lenguajes naturales cuentan con una colección de símbolos mucho más rica que el código ASCII. La realidad son palabras, coches, monitores, lamparas... Ninguno de ellos es fácilmente modelable a partir de un tipo derivado.

La respuesta a esta necesidad fue crear un nuevo tipo mucho más potente que puede contener dos familias de elementos: datos y métodos. Los datos son representaciones a un nivel más cercano al ordenador del contenido de un objeto. Por ejemplo, para describir el objeto bolígrafo se tomarían los datos capuchón, punta, tubo y tinta y podrían ser simples cadenas de caracteres. También son datos los estados, por ejemplo: lleno, vacío, tapado, roto... Se trata de, al igual que con un tipo derivado, describir con tipos propios de un ordenador la realidad. Los métodos son acciones propias de este objeto que describen la *interacción* del mismo con su entorno o consigo mismo. Métodos asociados al bolígrafo podrían ser destapar, escribir, morder o voltear.

La verdadera potencia de la orientación a objetos no es la de poder juntar datos y métodos en una misma construcción sino la capacidad de, mediante la herencia, crear nuevos objetos que modelen realidades más complejas. Por ejemplo, para modelar una pluma estilográfica junto con el bolígrafo pueden seguirse dos estrategias, la primera es utilizar bolígrafo como clase padre y *sobreescribir* el dato punta por plumilla. Otra opción sería crear un objeto padre de ambos, bolígrafo y pluma estilográfica, llamado herramienta de escritura con los datos y métodos comunes. Bolígrafo y pluma heredarían de herramienta de escritura y la ampliarían con los datos y métodos necesarios.

En Python los objetos se definen a partir de clases, siendo esta la la expresión más común entre los lenguajes de programación para la ejecución de la programación orientada a objetos. Una clase crea un tipo, tal como puede ser un entero, una lista o un diccionario; con la particularidad que ha sido formulada desde el inicio por el programador. Cuando se asigna una clase a una variable esta no contiene un objeto sino una *instancia* de la clase. De este modo se deja el vocablo objeto para un uso mucho más general. En Python las clases se definen de la siguiente manera:

```
In [1]: import math as m

In [2]: class Complex(object):
   ...:     def __init__(self, realp, imagp):
   ...:         self.r = realp
   ...:         self.i = imagp
   ...:
   ...:     def abs(self):
   ...:         return m.sqrt(self.r**2+self.i**2)
   ...:
```

Ya se ha definido una clase mínima que modela un número complejo. La función `__init__` es un método especial que describe la función a ejecutar durante la *instanciación* de la clase. Esto significa que al ejecutarse la clase `Complex` tendrá que hacerse con dos argumentos, la parte real y la parte imaginaria del número. `self` es una palabra clave que sirve para referirse a la clase en sí y se pasa como argumento para reforzar la consistencia sintáctica de Python. Cada método puede utilizar los datos definidos en la propia clase, como `self` define a la clase estos datos podrán utilizarse a partir de la clase como raíz tal como se aprecia en `self.r` y `self.i`. Sucede exactamente lo mismo con los métodos. Se denota la

accesibilidad de la propia clase en el método pasando `self` como argumento.

```
In [3]: nc = Complex(3,4)#instancia de clase

In [4]: nc.r
Out[4]: 3

In [5]: nc.i
Out[5]: 4

In [6]: nc.abs()
Out[6]: 5.0
```

Una vez creada la instancia del método ya puede accederse a sus datos y sus métodos. La clase `Complex` hereda de la clase `object` lo que no es más que un abuso de notación para explicitar que se trata de una clase que modela un objeto. A partir de la clase `Complex` se puede crear una clase `MoreComplex` que le añade un método a su progenitora:

```
class MoreComplex(Complex):
    def __init__(self, realp, imagp):
        Complex.__init__(self,realp,imagp)

    def arg(self):
        return m.atan(self.i/self.r)

In [7]: mc = MoreComplex(3,4)

In [8]: mc.arg()
Out[8]: 0.78539816339744828
```

De todas las posibles implementaciones prácticas de la orientación a objetos la elección de Python es el *duck typing*. Se describe mediante una frase bastante curiosa: *Si algo anda como un pato y cuaquea como un pato yo lo llamaré pato*. Esta elección es consistente con el hecho que en las funciones no es exige declarar el tipo de los argumentos. Si no se exige para los enteros o las cadenas de caracteres los objetos no deben ser una excepción[6]. Esto significa que si se pasa una instancia de una clase como argumento de una función ésta no va a realizar ninguna comprobación sobre la verdadera naturaleza de esta instancia. Se entenderá mejor este concepto con un ejemplo. Se define la clase *Pato*[7] que contiene un dato y un par de métodos. Uno de ellos es `haz_cua` que simplemente imprime `cua!` por pantalla.

```
>>> class pato:
...     cantidad = 1
...     def haz_cua(self):
...           print "cua!"
...
...     def reproducete(self):
...           cantidad += 1
...
>>> estoesunpato=pato() #instancia de pato
>>> estoesunpato.cantidad
1
>>> estoesunpato.haz_cua()
cua!
```

[6] Podemos recordar ahora la importancia del *zen de Python*

[7] Explicitar la herencia de la clase `object` es estrictamente opcional, si no existe herencia alguna proveniente de otras clases debe omitirse el paréntesis. Se deja al lector comprobar la inconsistencia sintáctica de dejar el paréntesis vacío.

El paso siguiente es crear una función, en este caso `cuaqueador` que pregunte a un supuesto pato si lo es realmente pidiéndole que cuaquee.

```
>>> estoesunpato=pato() #instancia de pato
>>> def cuaqueador(supuestopato):
...     supuestopato.haz_cua()
...
>>> cuaqueador(estoesunpato)
cua!
```

Efectivamente, como el objeto `pato` dispone de la función `haz_cua` no se produce ningún error. Pero yo soy perfectamente capaz de decir *cua!* si me piden que lo haga.

```
>>> class guillem:
...     def haz_cua(self):
...             print "cua!"
...
>>> falsopato=guillem() #ese soy yo
>>> cuaqueador(falsopato)
cua!
```

Evidentemente yo no soy un pato como puede comprobarse mediante la función `isinstance`

```
>>> isinstance(falsopato,pato)
False
```

El *duck typing* añade el máximo dinamismo posible a un lenguaje de programación pero puede también provocar grandes confusiones porque no existe ninguna imposición a nivel de interfaces entre objetos. Existen módulos que fuerzan la declaración de las clases cuando un argumento es un objeto, un ejemplo de ello es la clase `Interface` del popular entorno Zope3.

4. Numpy

Una de las primeras extensiones que se programaron para Python, quizás por ser una de las más necesarias, fue la que proporcionaba la clase de *array n-dimensional*. Un array es un tipo que puede describirse mediante dos características:

- Sus dimensiones
- El tipo de sus elementos

Esto implica que un *array*, a diferencia de las listas o los tuples, es homogéneo en memoria.
Un *array* es gracias al módulo Numpy otra clase de Python:

```
>>> from numpy import array
>>> help(array)

Help on built-in function array in module numpy.core.multiarray:

array(...)
    array(object, dtype=None, copy=1,order=None, subok=0,ndmin=0)

    Return an array from object with the specified date-type.

    Inputs:
```

```
      object  - an array, any object exposing the array interface, any
                object whose __array__ method returns an array, or any
                (nested) sequence.
      dtype   - The desired data-type for the array.  If not given,
                then the type will be determined as the minimum type
                required
   [...]
```

Junto con esta clase Numpy proporciona una colección mínima pero útil de funciones orientadas al cálculo numérico, más concretamente Transformadas Rápidas de Fourier y Álgebra Lineal.

La clase *array* en Python cuenta con más de ciento cincuenta métodos con, probablemente, todas las operaciones y acciones que puedan resultar de utilidad. Si se comparara con Matlab sin duda Python ganaría en el apartado de la versatilidad.

Para empezar a jugar con Numpy y los arrays lo mejor es utilizar el atajo siguiente:

```
>>> from numpy import *
```

Este comando carga las funciones usuales cuyos nombres en muchos casos provienen directamente de Matlab. He aquí una breve sesión como ejemplo:

```
>>> arange(5) #igual que range pero devuelve un array
array([0, 1, 2, 3, 4])
>>> array([0,1,2,3,4],'d') # array de doubles a partir de una lista
array([ 0.,  1.,  2.,  3.,  4.])
>>> array([0,1,2,3,4],'f') # array de floats a partir de una lista
array([ 0.,  1.,  2.,  3.,  4.], dtype=float32)
>>> array([0,1,2,3,4],dtype=complex64) # array de numeros complejos
array([ 0.+0.j,  1.+0.j,  2.+0.j,  3.+0.j,  4.+0.j], dtype=complex64)
>>> x = array([[1,2,3],[4,5,6],[7,8,9]],'d')
```

Como puede apreciarse en el último ejemplo el método de introducción de arrays es encapsulado, es decir, el índice que se encuentra en un nivel más interior será el primero. No existe el concepto de fila y columna como en Matlab.

```
>>> x = array([[1,2,3],[4,5,6],[7,8,9]],'d')
>>> y = array([1,0,0],'d')
>>> dot(x,y)
array([ 1.,  4.,  7.])
```

Como se puede apreciar se ha operado según el orden establecido de índices:

$$x \cdot y = \sum_i x_{ij} y_i$$

donde i era el primer índice y j el segundo proporcionando una notación independiente de la geometría de la matriz y mucho más *tensorial*. Por ejemplo, si se invierte el orden de los factores se consigue el resultado esperado sin necesidad de trasponer el vector:

```
>>> dot(y,x)
array([ 1.,  2.,  3.])
```

4.1. Indexing y Slicing

Numpy hereda de Python una característica que puede resultar desconcertante si no se entienden bien los conceptos de indexing y slicing. Los arrays son indexables de la manera usual *teniendo en cuenta que en Python, como en C, los índices se cuentan empezando con el cero*:

```
>>> x[0,1]
2.0
>>> x[0,:]
array([ 1.,  2.,  3.])
>>> x[0]
array([ 1.,  2.,  3.])
```

Las diferencia se encuentra si queremos porciones (*slices*) de la matriz:

```
>>> x[0:3,0:3]
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.],
       [ 7.,  8.,  9.]])
```

Intuitivamente y teniendo en cuenta tal como indexa Python la secuencia debería ir de cero a dos y no de cero a tres. Esto es así porque Python utiliza un criterio distinto para los índices y para las porciones. Se esquematiza perfectamente en esta figura:

El motivo de esta elección es que Python soporta los índices negativos, aunque no cíclicos —no puede pasarse de una frontera a otra—.

```
>>> x[-2:,-2:]
array([[ 5.,  6.],
       [ 8.,  9.]])
>>> x[-2:1,-2:1]
array([], shape=(0, 0), dtype=float64)
```

5. Scipy

Al poco de su aparición se constató que por su naturaleza Python podía ser un lenguaje adecuado para aplicarse a ciencia e ingeniería. Muchos lo vieron como un lenguaje con una filosofía parecida a Matlab pero que no contaba con ninguno de sus inconvenientes. El precio que había que pagar era que no existía ninguna biblioteca de funciones y utilidades parecida a la proporcionada por Matlab.

Afortunadamente el núcleo fundamental de Matlab, como el álgebra lineal o las transformadas de Fourier, es software libre. No hay ninguna diferencia esencial entre Matlab y su competencia a parte de los millares de pequeñas funciones que implementan pequeños algoritmos. Dotar a otro lenguaje de programación de la misma biblioteca de funciones es una tarea titánica pero técnicamente posible.

Scipy es el un esfuerzo común para dotar a Python, que significa respecto a Matlab un gran avance como lenguaje de programación, de una biblioteca de funciones equivalente. Se constata que el espejo es Matlab en cuanto se advierte que muchas de las funciones de Scipy toman el mismo nombre.

Scipy es una biblioteca basada en submódulos temáticos:

```
Available subpackages
---------------------
ndimage          --- n-dimensional image package [*]
stats            --- Statistical Functions [*]
signal           --- Signal Processing Tools [*]
lib              --- Python wrappers to external libraries [*]
linalg           --- Linear algebra routines [*]
linsolve.umfpack --- Interface to the UMFPACK library. [*]
odr              --- Orthogonal Distance Regression [*]
misc             --- Various utilities that don't have another home.
sparse           --- Sparse matrix [*]
interpolate      --- Interpolation Tools [*]
```

```
optimize          --- Optimization Tools [*]
cluster           --- Vector Quantization / Kmeans [*]
linsolve          --- Linear Solvers [*]
fftpack           --- Discrete Fourier Transform algorithms [*]
io                --- Data input and output [*]
maxentropy        --- Routines for fitting maximum entropy models [*]
integrate         --- Integration routines [*]
lib.lapack        --- Wrappers to LAPACK library [*]
special           --- Airy Functions [*]
lib.blas          --- Wrappers to BLAS library [*]
  [*] - using a package requires explicit import (see pkgload)
```

Scipy está enteramente basada en Numpy que nació para solucionar un pequeño gran problema para Python: la existencia de dos tipos de arrays.

Jim Hugunin desarrolló Numeric, la primera implementación de arrays para Python en 1995 cuando era estudiante del MIT. Numeric estaba diseñado para ser rápido y obviaba muchas de las posibles necesidades de la clase array dentro de un entorno interactivo. Para paliar estos pequeños inconvenientes Perry Greenfield, Todd Miller y Rick White crearon numarray pero nunca consiguieron la misma velocidad de ejecución que Numeric.

Esto provocó una gran incertidumbre dentro de la comunidad científica que se acercaba a Python. La tendencia era pensar que numarray se consolidaría y apartaría Numeric paulatinamente pero eso nunca llegaba a suceder porque era mucho más lento. Mientras para un programador era se trataba sólo de un dilema, para la creación de una gran biblioteca científica era sencillamente una herida sangrante.

Travis Oliphant se tomó el problema como algo personal y juntó lo mejor de ambas implementaciones en Numpy dando finalmente la posibilidad a Scipy y a Python de crecer dentro de la comunidad científica y técnica.

Ejercicio:

El gran punto débil de Scipy y por extensión, de Numpy, es la documentación. Para saber cómo utilizar una determinada función es más efectivo utilizar la ayuda interactiva del intérprete que buscar por internet porque no existe ningún directorio. Tampoco se proporciona ninguna base de datos de funciones con Scipy, algo que sería enormemente útil.
Afortunadamente parece que se está trabajando duramente en ello. Parece que la política será escribir sólo una documentación que pueda extraerse y moverse a distintos formatos gracias a un gestor de documentación llamado *sphinx*. Este ejercico pretende que el lector batalle un poco con la documentación resolviendo un problema de mínimos cuadrados generalizados.
Dentro del módulo de scipy *optimize* se encuentra la función `leastsq` que implementa el algoritmo Levenberg-Marqardt de mínimos cuadrados generalizados. Se trata de crear una serie de datos y ajustarlos mediante la función objetivo

$$f(x,\mathbf{p}) = \frac{1}{p_1 + p_2 e^{-x}}$$

Entonces el problema será minimizar, para una serie de datos (x_i, y_i) el funcional

$$R(\mathbf{p}) = \sum_i (y_i - f(x_i,\mathbf{p}))^2$$

Para crear los datos se tomará la función objetivo con $p_1 = 1$ y $p_2 = 2$, y se generará una serie de cien puntos sumando a la función un ruido aleatorio [A.2].
Una vez se ajusten los datos obtenidos mediante esta función el resultado dería ser precisamente $p_1 = 1$ y $p_2 = 2$.

A. Utilidades y soluciones a los problemas

A.1. Uso de los list comprehension

La solución al problema mediante un list comprehension en una sola línea es la siguiente:

```
>>> a=['uno','dos','tres','cuatro']
>>> b=[w for w in a if 'o' in w]
>>> b
['uno', 'dos', 'cuatro']
```

Esta sintaxis tiene sentido si se lee como si de una frase se tratara: *la palabra para cada palabra en a si 'o' está en la palabra*.

A.2. Generador de datos para el ejercicio de mínimos cuadrados generalizados

Este documento contiene el archivo `gendata.py` que contiene la función que genera los datos del ejercicio. Esta función devuelve un tuple de dos elementos y debe usarse como sigue:

```
>>> from gendata import gendata
>>> (x,y)=gendata()
```

B. Python 3.x

Esta sección deberá eliminarse en cuando la siguiente versión de Python, la 3.x se haya consolidado

Python es en la actualidad un lenguaje en plena transición a una nueva versión. El gran objetivo de Python 3.x, anteriormente conocido como Python 3000, es eliminar ciertos errores cometidos en los primeros estadios del desarrollo del lenguaje y que se han mantenido en él por la voluntad de no romper todo el código escrito. El objetivo secundario es conseguir ahondar en la propia filosofía de Python como lenguaje sencillo, consistente y corto.

A partir de la llegada de Python 3.x el código que ejecute en Python 2.x debe escribirse teniendo en cuenta muy seriamente los avisos del intérprete en tiempo de ejecución. En ellos se detallan problemas que pueden aparecer en un futuro proceso de migración.

Es también una demostración de valentía. Python es ya un lenguaje maduro y este cambio puede romper infinidad de código que hoy funciona sin ningún problema. Lejos de ser algo que concierne sólo a unos pocos desarrolladores sitios como Google, YouTube, NASA y miles de empresas, sitios o proyectos de software tendrán que dedicar algo de tiempo —esperemos que no mucho— al cambio de versión.

Referencias

[TUT] Python Tutorial; Guido van Rossum, Python Software Foundation; `http://docs.python.org/tut/tut.html`

[REF] Python Reference Manual; Guido van Rossum, Python Software Foundation; `http://docs.python.org/ref/ref.html`

[STB] Python Library Reference; Guido van Rossum, Python Software Foundation; `http://docs.python.org/lib/lib.html`

[OCT] Introducción Informal a Matlab y Octave; Guillem Borrell i Nogueras; `http://iimyo.forja.rediris.es`. ISBN: 978-8-4691-3626-3

[NUM] Guide to NumPy; Travis E. Oliphant; March 15, 2006; http://www.trelgol.com/