

## Chapter 2

# Introduction to Jython

In this chapter, we will give a short introduction to the Jython programming language. We have already pointed out that Jython is an implementation of the Python programming language, unlike CPython which is implemented in C/C++.

While these implementations provide almost identical a Python-language programming environment, there are several differences. Since Jython is fully implemented in Java, it is completely integrated into the Java platform, so one can call any Java class and method using the Python-language syntax. This has some consequences for the way you would program in Jython. During the execution of Jython programs, the Jython source code is translated to Java bytecode that can run on any computer that supports the Java virtual machine.

We cannot give a comprehensive overview of Jython or Python in this chapter: This chapter aims to describe a bare minimum which is necessary to understand the Jython language, and to provide the reader with sufficient information for the following chapters describing data-analysis techniques using the jHepWork libraries.

## 2.1 Code Structure and Commentary

As for CPython, Jython programs can be put into usual text files with the extension `.py`. A Jython code is a sequence of statements that can be executed normally, line-by-line, from the top to the bottom. Jython statements can also be executed interactively using the Jython shell (the tab `JythonShell` of the jHepWork IDE).

Comments inside Jython programs can be included using two methods: (1) To make a single-line comment, put the sharp `"#"` at the beginning of the line; (2) To comment out a multi-line block of a code, use a triple-quoted string.

It is good idea to document each piece of the code you are writing. Documentation comments are strings positioned immediately after the start of a module, class

or function. Such comment can be accessed via the special attribute `__doc__`. This attribute will be considered later on when we will discuss functions and classes.

Jython statements can span multiple lines. In such cases, you should add a back slash at the end of the previous line to indicate that you are continuing on the next line.

## 2.2 Quick Introduction to Jython Objects

As for any dynamically-typed high-level computational language, one can use Jython as a simple calculator. Let us use the jHepWork IDE to illustrate this. Start the jHepWork and click on the “JythonShell” tab panel. You will see the Jython invitation “>>>” to type a command. Let us type the following expression:

```
>>> 100*3/15
```

Press [Enter]. The prompt returns “20” as you would expect for the expression  $(100 \cdot 3) / 15$ . There was no any assignment in this expression, so Jython assumes that you just want to see the output. Now, try to assign this expression to some variable, say `W`:

```
>>> W=100*3/15
```

This time, no any output will be printed, since the output of the expression from the right side is assigned directly to the variable `W`. Jython supports multiple assignments, which can be rather handy to keep the code short. Below we define three variables `W1`, `W2` and `W3`, assigning them to 0 value:

```
>>> W1=W2=W3=0
```

One can also use the parallel assignments using a sequence of values. In the example below we make the assignment `W1=1`, `W2=2` and `W3=3` as:

```
>>> W1,W2,W3=1,2,3
```

At any step of your code, you can check which names in your program are defined. Use the built-in function `dir()` which returns a sorted list of strings

```
>>> dir()
['W', 'W1', 'W2', 'W3', '__doc__', '__name__' ...]
```

(we have truncated the output in this example since the actual output is rather long). So, Jython knows about all our defined variables including that defined using the character `W`. You will see more variables in this printed list which are predefined by `jHepWork`.

One can print out variables with the `print()` method as:

```
>>> print W1
1
```

One can also append a comment in front:

```
>>> print 'The output =',W1
The output = 1
```

You may notice that there is a comma in front the variable `W1`. This is because `'The output'` is a string, while `'W1'` is an integer value, so their types are distinct and must be separated in the print statement.

How do we know the variable types? For this we can use the `type()` method which determines the object type:

```
>>> type(W1)
<type 'int'>
```

The output tells that this variable holds an integer value (the type `"int"`).

Let us continue with this example by introducing another variable, `'S'` and by assigning a text message to it. The type of this variable is `"str"` (string).

```
>>> S='The output ='
>>> type (S)
<type 'str'>
```

So, the types of the variables `'W1'` and `S` are different. This illustrates the fact that Jython, as any high-level language, determines types based on assigned values during execution, i.e. a variable may hold a binding to any kind of object. This feature is very powerful and the most useful for scripting: now we do not need to worry about defining variable types before making assignments to a variable. Clearly, this significantly simplifies program development.

Yet, the mechanics behind such useful feature is not so simple: the price to pay for such dynamical features is that all variables, even such simple as `'W1'` and `'S'` defined before, are objects. Thus, they are more complicated than simple types in other programming languages, like `C/C++` or `FORTRAN`. The price to pay is slower execution and larger memory consumption. On the other hand, this also means that you can do a lot using such feature! It should also be noted that some people use

the word “value” when they talk about simple types, such as numbers and strings. This is because these objects cannot be changed after creation, i.e. they are immutable.

First, let us find out what can we do with the object ‘w1’. We know that it holds the value 10 (and can hold any value). To find out what can be done with any objects in the JythonShell window is rather simple: Type ‘w1’ followed by a dot and press the [Space] key by holding down [Ctrl]:

```
>>> w1. [Ctrl]-[Space]
```

You will see a list of methods attributed to this object. They usually start as `__method__`, where ‘method’ is some attribute. For example, you will see the method like `__str__`, which transforms an object of type integer to a string. So, try this

```
>>> SW=str(w1); type(SW)
<type 'str'>
```

Here we put two Jython statements on one line separated by a semi-column. This, probably, is not very popular way for programming in Jython, but we use it to illustrate that one can this syntax is also possible. In some cases, however, a program readability can significantly benefit from this style if a code contains many similar and short statements, such as `a1=1; a2=2`. In this case, the statements have certain similarity and it is better to keep them in one single logical unit. In addition, we will use this style in several examples to make our code snippets short.

The last expression in the above line does not have “=”, so Jython assumes that what you really want is to redirect the output to the interactive prompt. The method `type()` tells that “SW” is a string. As before, you may again look at the methods of this object as:

```
>>> SW. [Ctrl]-[Space]
```

This displays a list of the methods attributed to this object. One can select a necessary method and insert it right after the dot.

In addition to the JythonShell help system, one can discover the attributes of each Jython object using the native Jython method `dir()`:

```
>>> dir(SW)
```

In the following sections we will discuss other methods useful to discover attributes of Java objects.

### 2.2.1 Numbers as Objects

Numbers in Jython are immutable objects called values, rather than simple types as in other programming languages (C/C++, Fortran or Java). There are two main types: integers (no fractional part) and floats (with fractional part). Integers can be represented by long values if they are followed by the symbol ‘L’. Try this:

```
>>> Long=20L
<type 'long'>
```

The only limit in the representation of the long numbers is the memory of Java virtual machine.

Let us take a look again at the methods of a real number, say “20.2” (without any assignment to a variable).

```
>> 20.2. [Ctrl]-[Space]
```

or, better, you can print them using the method “dir()”

```
>>> dir(20.2)
```

Again, since there is no any assignment, Jython just prints the output of the dir() method directly to the same prompt. Why they are needed and what you can do with them? Some of them are rather obvious and shown in Table 2.1.

**Table 2.1** A short overview of the Jython operators for values

Jython operators for values		
<i>abs(x)</i>	<code>__abs__</code>	absolute value
<i>pow(x, y)</i> or <i>y**x</i>	<code>__pow__</code>	raise <i>x</i> to the power <i>y</i>
<i>-x, +x</i>	<code>__neg__</code> , <code>__pos__</code>	negative or positive
<i>+, -</i>	<code>__radd__</code> , <code>__rsub__</code>	add or subtract
<i>*, /</i>	<code>__rmul__</code> , <code>__rdiv__</code>	add or subtract
<i>x &lt; y, x &gt; y</i>	<code>__com__</code>	less or larger. Returns 0 (false) or 1 (true)
<i>cmp(x, y)</i>	<code>__com__</code>	compare numbers. Returns 0 (false) or 1 (true)
<i>x &lt;= y, x &gt;= y</i>	<code>-</code>	comparison: less (greater) or equal
<i>x == y, x != y</i>	<code>-</code>	comparison: equal or not equal
<i>str(x)</i>	<code>__str__</code>	convert to a string
<i>float(x)</i>	<code>__float__</code>	convert to float
<i>int(x)</i>	<code>__int__</code>	convert to integer
<i>long(x)</i>	<code>__long__</code>	convert to long

There are more methods designed for this object, but we will not go into further discussion. Just to recall: any number in Jython is an object and you can manipulate with it as with any object to be discussed below. For example, Jython integers are objects holding integer values. This is unlike C++ and Java where integers are primitive types.

Is it good if simple entities, such as numbers are, have properties of objects? For interactive manipulation with a code and fast prototyping, probably we do not care so much, or even can take advantage of this property. But, for numerical libraries, this feature is unnecessary and, certainly, is too heavy for high-performance calculations. We will address this issue later in the text.

### 2.2.2 Formatted Output

In the above examples we have used the `print` command without setting control over the way in which printed values are displayed. For example, in the case of the expression `"1.0/3.0"`, Jython prints the answer with 17 digits after the decimal place!

Obviously, as for any programming language, one can control the way the values are displayed: For this, one can use the `%` command to produce a nicely formatted output. This is especially important if one needs to control the number of decimal places the number is printed to. For example, one can print `1.0/3.0` to three decimal places using the operator `%.3f` inside the string:

```
>>> print 'The answer is %.3f'%(1.0/3)
The answer is 0.333
```

As you can see, Jython replaces the character `"f"` with the variable value that follows the string. One can print more than one variable as shown in the example below:

```
>>> print 'The answer is %.3f and %.1f'%(1.0/3, 2.0/3)
The answer is 0.333 and 0.7
```

One can also use the operator `%` to control the width of the displayed number, so one can make neatly aligned tables. For example, the string `"10.1f"` forces the number to be printed such that it takes up to ten characters. The example below shows how to do this using the new-line character to print the second number on a new line. As one can see, we align this second number with that printed on the first line:

```
>>> print 'The answer: %.3f \n %13.1f'%(1.0/3, 2.0/3)
The answer: 0.333
              0.7
```

### 2.2.3 Mathematical Functions

To perform mathematical calculations with values, one should use the Jython `math` module which comes from the standard specification of the Python programming language. Let us take a look at what is inside of this module. First, we have to import this module using the “`import math`” statement:

```
>>> import math
```

Use the usual approach to find the methods of this module:

```
>>> dir(math)
['acos', 'asin', 'atan', 'atan2', 'ceil',
 'classDictInit', 'cos', 'cosh', 'e', 'exp',
 'fabs', 'floor', 'fmod', 'frexp', 'hypot',
 'ldexp', 'log', 'log10', 'modf', 'pi', 'pow',
 'sin', 'sinh', 'sqrt', 'tan', 'tanh']
```

Most of us are familiar with all these mathematical functions that have the same names in any programming language. To use these functions, type the module name `math` followed by the function name. A dot must be inserted to separate the module and the function name:

```
>>> math.sqrt(20)
4.47213595499958
```

As before for JythonShell, one can pick up a necessary function as:

```
>>> math. [Ctrl]-[Space]
```

It should be noted that, besides functions, the `math` module includes a few well-known constants:  $\pi$  and  $e$ :

```
>>> print "PI=", math.pi
PI= 3.141592653589793
>>> print "e=", math.e
e= 2.718281828459045
```

If you have many mathematical operations and want to make a code shorter by skipping the “`math`” attribute in front of each function declaration, one can explicitly import all mathematical functions using the symbol “`*`”:

```
>>> from math import *  
>>> sqrt(20)  
4.47213595499958
```

### 2.2.4 Complex Numbers

Python has a natural support for complex numbers. Just attach “J” or “j” for the imaginary part of a complex number:

```
>>> C=2+3j  
>>> type(C)  
<type 'complex'>
```

Once a complex number is defined, one can perform mathematical manipulations as with the usual numbers. For example:

```
>>> 1j*1j  
(-1+0j)
```

Mathematical operations with complex numbers can be performed using the ‘`cmath`’ module, which is analogous to the ‘`math`’ module discussed above. The example below demonstrates how to calculate hyperbolic cosine of a complex value:

```
>>> import cmath  
>>> print cmath.cosh( 2+3j )  
(-3.724+0.511j)
```

The output values for the real and imaginary part in the above example were truncated to fit the page width.

## 2.3 Strings as Objects

Strings are also treated as values since they are immutable. To define a string, one should enclose it in double (") or single (') quote. The escape character is a backslash, so one can put a quote character after it. The newline is given by *n* directly after the backslash character. Two strings can be added together using the usual “+” operator.

As mentioned above, an arbitrary value, `val`, can be converted into a string using the method `str(val)`. To convert a string into `int` or `float` value, use the

methods `int(str)` or `float(str)`. Below we illustrate several such conversions:

```
>>> i=int('20')
>>> type(i)
<type 'int'>
>>> f=float('20.5')
>>> type(f)
<type 'float'>
```

As before, all the methods associated with a string can be found using [Ctrl]-[Space] or the `dir()` method:

```
>>> dir('s')
...
'capitalize', 'center', 'count', 'decode',
'encode', 'endswith', 'expandtabs', 'find',
'index', 'isalnum', 'isalpha', 'isdecimal',
'isdigit', 'islower', 'isnumeric', 'isspace'...
```

(we display only a few first methods). Some methods are rather obvious and do not require explanation. All methods that start from the string “is” check for a particular string feature.

Below we list more methods:

<code>len(str)</code>	gives the number of characters in the string <code>str</code>
<code>string.count(str)</code>	counts the number of times a given word appears in a string
<code>string.find(str)</code>	numeric position of the first occurrence of word in the string
<code>str.lower()</code>	returns a string with all lower case letters
<code>str.upper()</code>	returns a string with all upper case letters

Strings can be compared using the standard operators: `==`, `!=`, `<`, `>`, `<=`, and `>=`.

## 2.4 Import Statements

There are several ways that can be used to import a Java or Python package. One can use the `'import'` statement followed by the package name. In case of Python, this corresponds to a file name without the extension `'.py'`. The import statement executes the imported file, unlike lower-level languages, like C/C++, where the import statement is a preprocessor statement. The consequence of this is that the import statement can be located in any place of your code, as for the usual executable statement. We have seen already how to import the Python package “math”.

Here is an example illustrating how to import the Java Swing package (usually used to build a GUI):

```
>>> from javax.swing import *
```

In the above example we use the wildcard character “\*” to import all packages from Java Swing. In this book, you will see that we use “\*” wildcard almost for every example, since we want to keep our examples short. This is often considered as a bad style since it “pollutes” the global namespace. However, if you know that the code is not going to be very long and complicated, we should not worry too much about this style.

Let us give another example showing how to import of a Java class. We remind that the code below works only for Python implemented in Java (Jython):

```
>>> from javax.swing import JFrame
```

This time we have imported only a single class (JFrame) from Java, unlike the previous example with the “polluted” namespace.

Another way to import classes is to use the ‘import’ statement without the string ‘from’. For example:

```
>>> import javax.swing
```

In this case, we should use the qualified names, i.e.:

```
>>> f=javax.swing.JFrame('Hello')
```

Although it takes more typing, we have avoided polluting the global namespace of our code.

### ***2.4.1 Executing Native Applications***

In Sect. 1.4.3 we have shown that native applications can be run using JythonShell by appending “!” in front of an external command. In addition, one can also use Jython ‘os.system’ package to run an external program.

The code below shows how to run an external command. In this example, we bring up the Acroread PDF file viewer (it should be found on the current PATH if this program installed on your system):

```
>>> import os
>>> rc=os.system('acroread')
```

```
>>> if rc == 0:  
>>> ... print 'acoread started successfully'
```

The statement 'if' checks whether the execution has succeeded or not. We will discuss the comparison tests in the next section. The same operation will look like !acoread when using the Jython shell.

## 2.5 Comparison Tests and Loops

### 2.5.1 The 'if-else' Statement

Obviously, as in any programming language, one can use the 'if-else' statement for decision capability of your code. The general structure of comparison tests is

```
if [condition1]:  
    [statements to execute if condition1 is true]  
elif [condition2]:  
    [statements to execute if condition2 is true]  
....  
else:  
    [rest of the program]
```

The text enclosed in square brackets represents some Jython code. After the line with the statement 'if', the code is placed farther to the right using white spaces in order to define the program block. Either space characters or tab characters (or even both!) are accepted as forms of indentation. In this book, we prefer two spaces for indentation. It should also be noted that the exact amount of indentation does not matter, only the relative indentation of nested blocks (relative to each other) is important.

The indentation is good Python feature: The language syntax forces to use the indentation that you would have used anyway to make your program readable. Thus even a lousy programmer is forced to write understandable code!

Now let us come back to the comparison tests. The [condition] statement has several possibilities for values 'a' and 'b' values as shown in Table 2.2:

Let us illustrate this in the example below:

```
>>> a=1; b=2;  
>>> if a*b>1:  
>>> .. print "a*b>1"  
>>> else:  
>>> .. print "a*b<=1"
```

**Table 2.2** Most popular Jython comparison tests

Comparison tests	
$a == b$	a is equal to b
$a != b$	a is not equal to b
$a > b$	a is greater than b
$a >= b$	a is greater than or equal to b
$a < b$	a is less than b
$a <= b$	a is less than or equal to b
$a == b$	a is equal to b
$a != b$	a is not equal to b

In case if you will need more complex comparisons, use the boolean operators such as 'and' and 'or':

```
>>> a=1; b=0
>>> if a>0 and b=0:
>>> ..print 'it works!'
```

One can also use the string comparisons = (equal) or != (not equal). The comparison statements are case sensitive, i.e. 'a' == 'A' is false.

### 2.5.2 Loops. The “for” Statement

The need to repeat a statement or a code block is essential feature of any numerical calculation. There is, however, one feature you should be aware of: Python should be viewed as an “interface” type of language, rather than that used for heavy repeated operations like long loops over values. According to the author’s experience, if the number of iterations involving looping over values is larger than several thousands, such part of the code should be moved to an external library to achieve a higher performance and a lower memory usage compared to the Python code operating with loops over objects. In case of Java, such libraries should be written in Java.

In this section we will be rather short. One can find more detailed discussion about this topic in any Python textbook.

The simplest loop which prints, say, 10 numbers is shown below:

```
>>> for i in range(10):
>>> ... print i
```

This 'for' loop iterates from 0 to 9. Generally, you can increment the counter by any number. For example, to print numbers from 4 to 10 with the step 2, use this example:

```
>>> for i in range(4,10,2):  
>>> ... print i
```

### 2.5.3 The ‘*continue*’ and ‘*break*’ Statements

The loops can always be terminated using the ‘*break*’ statement, or some iterations can be skipped using the ‘*continue*’ statement. All such control statements are rather convenient, since help to avoid various ‘*if*’ statements which makes the Python code difficult to understand. This is illustrated in the example bellow:

```
>>> for i in range(10):  
>>> ... if (i == 4): continue  
>>> ... if (i == 8): break  
>>> ... print i
```

In this loop, we skip the number 6 and break the loop after the number 8:

### 2.5.4 Loops. The ‘*while*’ Statement

One can also construct a loop using the ‘*while*’ statement, which is more flexible since its iteration condition could be more general. A generic form of such loop is shown below:

```
while CONDITION:  
... <Code Block as long as CONDITION is true>
```

Let us give a short example which illustrates the while loop:

```
>>> a=0  
>>> while a<10:  
>>> ... a=a+1
```

The while loop terminates when  $a=10$ , i.e. when the statement after the ‘*while*’ is false. As before, one can use the control statements discussed above to avoid overloading the execution block with various “*if*” statements.

One can also create an infinite loop and then terminate it using the “*break*” statement:

```
>>> a=0
>>> while 1:
>>> ... print "infinite loop!"
>>> ... a=a+1;
>>> ... if a>10:
>>>         break
>>> ... print i
```

In this example, the 'break' statement together with the 'if' condition controls the number of iterations.

## 2.6 Collections

Data-analysis computations are usually based on object collections, since they have being designed for various repetitious operations on sequential data structures—exactly what we mostly do when analyzing multiple measurements. In addition, a typical measurement consists of a set of observations which have to be stored in a data container as a single unit.

Unlike to other languages, we consider Python collections to be useful mainly for storing and manipulation with other high-level objects, such as collections with a better optimized performance for numerical calculations. In this book, we will use the Jython collections to store sets of jHepWork histograms, mathematical functions, Java-based data containers and so on.

Of course, one can use Jython collections to keep numerical values, but this approach is not going to be very efficient: An analysis of such values requires Python loops which are known to be slow. Secondly, there are no too many pre-build Jython libraries for object manipulation.

Nevertheless, in many parts of this books we will use collections which contain numerical values: this is mainly for pedagogical reasons. Besides, we do not care too much about the speed of our example programs when analyzing tens of thousands events.

### 2.6.1 Lists

As you may guess, a list is an object which holds other objects, including values. The list belongs to a *sequence*, i.e. an ordered collection of items.

## 2.6.2 List Creation

An empty list can be created using squared brackets. Let us create a list and check its methods:

```
>>> list=[]
>>> dir(list) # or list. + [Ctrl]+[Space]
```

One can also create a list which contains integer or float values during the initialization:

```
>>> list=[1,2,3,4]
>>> print list
[1, 2, 3, 4]
```

The size of this list is accessed using the `len(list)` method. The minimum and maximum values are given by the `min(list)` and `max(list)` methods, respectively. Finally, for a list which keeps numerical values, one can sum-up all list elements as `sum(list)`.

One can create a mixed list with numbers, strings or even other lists:

```
>>> list=[1.0,'test',int(3),long(2),[20,21,23]]
>>> print list
[1.0, 'test', 3, 2L, [20, 21, 23]]
```

One can obtain each element of the list as `list[i]`, where 'i' is the element index,  $0 < i < \text{len}(\text{list})$ . One can select a slice as `list[i1:i2]`, or even select the entire list as `list[:]`. A slice which selects index 0 through 'i' can be written as `list[:i]`. Several lists can be concatenated using the plus operator '+', or one can repeat the sequence inside a list using the multiplication '\* '.

As before, one can find the major methods of the list using `[Ctrl]+[Space]` keys. Some methods are rather obvious:

To add a new value, use the method `append()`:

```
>>> list.append('new string')
```

A typical approach to fill a list in a loop would be:

```
>>> list=[]
>>> for i in range(4,10,2):
>>> ...     list.append(i)
```

(here, we use a step 2 from 4 to 10). The same code in a more elegant form looks like:

```
>>> list=range(4, 10, 2)
>>> print list
[4, 6, 8]
```

If one needs a simple sequence, say from 0 to 9 with the step 1, this code can be simplified:

```
>>> list=range(10)
>>> print 'List from 0 to 9:',list
List from 0 to 9: [0,1,2,3,4,5,6,7,8,9]
```

One can create a list by adding some condition to the `range` statement. For example, one create lists with odd and even numbers:

```
>>> odd =range(1,10)[0::2]
>>> even=range(1,10)[1::2]
```

Another effective “one-line” approach to fill a list with values is demonstrated below:

```
>>> import math
>>> list = [math.sqrt(i) for i in range(10)]
```

Here we created a sequence of `sqrt(i)` numbers with  $i = 0..9$ .

Finally, one can use the ‘`while`’ statement for adding values in a loop. Below we make a list which contains ten zero values:

```
>>> list=[]
>>> while len(list)<10:
>>> ... list.append(0)
```

### 2.6.3 Iteration over Elements

Looping over a list can be done with the ‘`for`’ statement as:

```
>>> for i in list:
>>> ...print i
```

or calling its elements by their index ‘`i`’:

```
>>> for i in range(len(list)):
>>> ...print list[i]
```

### 2.6.3.1 Sorting, Searches, Removing Duplicates

The list can be sorted with the `sort()` method:

```
>>> list.sort()
>>> print list
[1.0, 2L, 3, [20, 21, 23], 'new string', 'test']
```

To reverse the list, use the method `reverse()`.

To insert a value, use the `insert(val)` method, while to remove an element, use the `remove(val)` method. Finally, one can delete either one element of a list or a slice of elements. For example, to remove one element with the index `i1` of a list use this line of the code: `'del list[i1]'`. To remove a slice of elements in the index range `i1-i2`, use `'del list[i1:i2]'`. To empty a list, use `'del list[:]'`. Finally, `'del list'` removes the list object from the computer memory.

It should be noted that the list size in the computer memory depends on the number of objects in the list, not on the size of objects, since the list contains pointers to the objects, not objects themselves.

Advanced statistical analysis will be considered in Sect. 7.4, where we will show how to access the mean values, median, standard deviations, moments and etc. of distributions represented by Jython lists.

Jython lists are directly mapped to the Java ordered collection `List`. For example, if a Java function returns `ArrayList<Double>`, this will be seen by Jython as a list with double values.

To search for a particular value `'val'`, use

```
>>> if val in list:
>>> ...print 'list contains', val
```

For searching values, use the method `index(val)`, which returns the index of the first matching value. To count the number of matched elements, the method `count(val)` can be used (it also returns an integer value).

### 2.6.4 Removal of Duplicates

Often, you may need to remove a duplicate element from a list. To perform this task, use the-called dictionary collection (will be discussed below). The example to

be given below assumes that a `list` object has been created before, and now we create a new list (with the same name) but without duplicates:

```
>>> tmp={}
>>> for x in list:
>>> ...tmp[x] = x
>>> list=tmp.values()
```

This is usually considered to be the fastest algorithm (and the shortest). However, this method works for the so-called hashable objects, i.e. class instances with a “hash” value which does not change during their lifetime. All Jython immutable built-in objects are hashable, while all mutable containers (such as lists or dictionaries to be discussed below) are not. Objects which are instances of user-defined Jython or Java classes are hashable.

For unhashable objects, one can first sort objects and then scan and compare them. In this case, a single pass is enough for duplicate removal:

```
>>> list.sort()
>>> last = list[-1]
>>> for i in range(len(list)-2, -1, -1):
>>> ...if last==list[i]:
>>>     del list[i]
>>> ...else:
>>>     last=list[i]
```

The code above is considered to be the second fastest method after that based on the dictionaries. The method above works for any type of elements inside lists.

### 2.6.4.1 Examples

Lists are very handy for many data-analysis applications. For example, one can keep names of input data files which can be processed by your program in a sequential order. Or, one can create a matrix of numbers for linear algebra. Below we will give two small examples relevant for data analysis:

*A matrix.* Let us create a simple matrix with integer or float numbers:

```
>>> mx=[
...   [1, 2],
...   [3, 4],
...   [5, 6],
...   ]
```

One can access a row of this matrix as `mx[i]`, where ‘i’ is a row index. One can swap rows with columns and then access a particular column as:

```
>>> col=[x[0] for x in mx], [x[1] for x in mx]]
>>> print col
[[1, 3, 5], [2, 4, 6]]
```

In case of an arbitrary number of rows in a matrix, use the map container for the same task:

```
>>> col=map(None,*mx)
>>> print col
[[1, 3, 5], [2, 4, 6]]
```

Advanced linear-algebra matrix operations using a pure Jython approach will be considered in Sect. 7.5.4.

*Records with measurements.* Now we will show that the lists are very flexible for storing records of data. In the example below we create three records that keep information about measurements characterized by some identification string, a time stamp indicating when the measurement is done and a list with actual numerical data:

```
>>> meas=[]
>>> meas.append(['test1', '06-08-2009', [1,2,3,4]])
>>> meas.append(['test2', '06-09-2009', [8,1,4,4,2]])
>>> meas.append(['test3', '06-10-2009', [9,3]])
```

This time we append lists with records to the list holding all event records. We may note that the actual numbers are stored in a separate list which can have an arbitrary length (and could also contain other lists). To access a particular record inside the list `meas` use its indexes:

```
>>> print meas[0]
>>> ['test1', '06-08-2009', [1, 2, 3, 4]]
>>> print meas[0][2]
[1, 2, 3, 4]
```

### 2.6.5 Tuples

Unlike lists, tuples cannot be changed after their creation, thus they cannot grow or shrink as the lists. Therefore, they are *immutable*, similar to the values. As the Jython lists, they can contain objects of any type. Tuples are very similar to the lists and can be initiated in a similar way:

```
>>> tup=() # empty tuple
>>> tup=(1,2,"test",20.0) # with 4 elements
```

Of course, now operations that can change the object (such as `append()`), cannot be applied, since we cannot change the size of this container.

In case if you need to convert a list to a tuple, use this method:

```
>>> tup=tuple([1,2,3,4,4])
```

Below we will discuss more advanced methods which add more features to manipulations with the lists and tuples.

### 2.6.6 Functional Programming. Operations with Lists

Functional programming in Jython allows to perform various operations on data structures, like lists or tuples. For example, to create a new list by applying the formula:

$$\frac{b[i] - a[i]}{b[i] + a[i]} \quad (2.1)$$

for each element of two lists, `a` and `b`, you would write a code such as:

```
>>> a=[1,2,3]
>>> b=[3,4,5]
>>> c=[]
>>> for i in range(len(a)):
>>> ... c.append( b[i]-a[i] / (a[i]+b[i]) )
```

To circumvent such unnecessary complexity, one can reduce this code to a single line using functional programming:

```
>>> a=[1.,2.,3.]
>>> b=[3.,4.,5.]
>>> c= map(lambda x,y: (y-x)/(y+x),a,b)
>>> print c
[0.5, 0.33, 0.25]
```

The function `map` creates a new list by applying (2.1) for each element of the input lists. The statement `lambda` creates a small anonymous function at runtime which tells what should be done with the input lists (we discuss this briefly in Sect. 2.10).

As you can see, the example contains much lesser code and, obviously, programming is done at a much higher level of abstraction than in the case with the usual loops over list elements.

To build a new list, one can also use the 'math' module. Let us show a rather practical example based on this module: assume we have made a set of measurements, and, in each measurement, we simply counting events with our observations. The statistical error for each measurement is the square root of the number of events, in case of counting experiments like this. Let us generate a list with statistical errors from the list with the numbers of events:

```
>>> data=[4,9,25,100]
>>> import math
>>> errors= map(lambda x: math.sqrt(x),data)
>>> print errors
[2.0, 3.0, 5.0, 10.0]
```

The above calculation requires one line of the code, excluding the standard 'import' statement and the 'print' command.

Yet, you may not be totally satisfied with the 'lambda' function: sometime one needs to create a rather complicated function operating on lists. Then one can use the standard Jython functions:

```
>>> a=[1.,2.,3.]
>>> b=[3.,4.,5.]
>>> def calc(x,y):
>>> ... return (y-x)/(x+y)
>>> c= map(calc,a,b)
>>> print c
[0.5, 0.33, 0.25]
```

The functionality of this code is totally identical to that of the previous example. But, this time, the function `calc()` is the so-called "named" Jython function. This function can contain rather complicated logic which may not fit to a single-line 'lambda' statement.

One can also create a new list by selecting certain elements. In this case, use the statement `filter()` which accepts an one-argument function. Such function must return the logical `true` if the element should be selected. In the example below we create a new list by taking only positive values:

```
>>> a=[-1,-2,0,1,2]
>>> print "filtered:",filter(lambda x: x>0, a)
filtered: [1, 2]
```

As before, the statement `'lambda'` may not be enough for more complicated logic for element selection. In this case, one can define an external (or named) function as in the example below:

```
>>> a=[-1,-2,0,1,2]
>>> def posi(x):
>>> ... return x > 0
>>> print "filtered:",filter(posi, a)
filtered: [1, 2]
```

Again the advantage of this approach is clear: we define a function `posi()`, which can arbitrarily be complicated, but the price to pay is more coding.

Finally, one can use the function `reduce()` that applies a certain function to each pair of items. The results are accumulated as shown below:

```
>>> print "accumulate:",reduce(lambda x, y: x+y,[1,2,3])
>>> accumulate: 6
```

The same functional programming methods can be applied to the tuples.

### 2.6.7 Dictionaries

Another very useful container for analysis of data is the so-called dictionary. If one needs to store some objects (which, in turn, could contain other objects, such as more efficiently organized collections of numbers), it would be rather good idea to annotate such elements. Or, at least, to have some human-readable description for each stored element, rather than using an index for accessing elements inside the container as for lists or tuples. Such a description, or the so-called “key”, can be used for fast element retrieval from a container.

Dictionaries in Jython (as in Python) are designed for one-to-one relationships between keys and values. The keys and the corresponding values can be any objects. In particular, the dictionary value can be a string, numerical value or even other collection, such as a list, a tuple, or other dictionary.

Let us give an example with two keys in form of strings, `'one'` and `'two'`, which map to the integer values `'1'` and `'2'`, respectively:

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

In this example, we have used the key `'one'` to access the integer value `'1'`. One can easily modify the value using the key:

```
>>> dic['one']=10
```

It should be noted that the keys cannot have duplicate values. Assigning a value to the existing key erases the old value. This feature was used when we removed duplicates from the list in Sect 2.6.3.1. In addition, dictionaries have no concept of order among elements.

One can print the available keys as:

```
>>> print dic.keys()
```

The easiest way to iterate over values would be to loop over the keys:

```
>>> for key in dic:  
>>> ... print key, 'corresponds to', dic[key]
```

Before going further, let us rewrite the measurement example given in the previous section when we discussed the lists. This time we will use record identifications as keys for fast retrieval:

```
>>> meas={}  
>>> meas['test1']=['06-08-2009', [1,2,3,4]]  
>>> meas['test2']=['06-09-2009', [8,1,4,4,2]]  
>>> meas['test3']=['06-10-2009', [9,3]]  
>>> print meas['test2']  
['06-09-2009', [8, 1, 4, 4, 2]]
```

In this case, one can quickly access the actual data records using the keys. In our example, a single data record is represented by a list with the date and additional list with numerical values.

Let us come back to the description of the dictionaries. Here are a few important methods we should know about:

```
dic.clear()   clean a dictionary;  
dic.copy()    make a copy;  
has_key(key)  test, is a key present?;  
keys()        returns a list of keys;  
values()      returns a list of values in the dictionary.
```

One can delete entries from a dictionary in the same way as for the list:

```
>>> del dic['one']
```

One can sort the dictionary keys using the following approach: convert them into a list and use the `sort()` method for sorting:

```
>>> people = {'Eve':10, 'Tom': 20, 'Arnold': 50}
>>> list = people.keys()
>>> list.sort()
>>> for p in list:
>>> ... print p, 'is ', people[p]
Arnold is 50
Eve is 10
Tom is 20
```

## 2.7 Java Collections in Jython

It was already said that the concept of collections is very important for any data analysis, since “packing” multiple records with information into a single unit is a very common task.

There are many situations when it is imperative to go beyond the standard Python-type collections implemented in Jython. The strength of Jython is in its complete integration with Java, thus one can call Java collections to store data. Yes, the power of Java is in your hands!

To access Java collections, first you need to import the classes from the package `java.util`. Java collections usually have the class names started with capital letters, since this is the standard convention for class names in the Java programming language. With this observation in mind, there is a little chance for mixing Python collections with Java classes during the code development. In this section, we will consider several collections from the Java platform.

### 2.7.1 List. An Ordered Collection

To build an ordered collection which contain duplicates, use the class `List` from the Java package `java.util`. Since we are talking about Java, one can check what is inside of this Java package as:

```
>>> from java.util import *
>>> dir()
[ .. 'ArrayList', 'Currency', 'Date', 'List', 'Set', 'Map']
```

Here we printed only a few Java classes to fit the long list of classes to the page width. One can easily identify the class `ArrayList`, a class which is usually used

to keep elements in a list. One can check the type of this class and its methods using either `dir()` or the JythonShell code assist:

```
>>> from java.util import *
>>> jlist=ArrayList()
>>> type(jlist)
<type 'java.util.ArrayList'>
>>> dir(jlist):
[... methods ...]
>>> jlist. # [Ctrl]+[Space]
```

As you can see, the `type()` method indicates that this is a Java instance, so we have to use the Java methods of this instance for further manipulation. Let us add elements to this list and print them:

```
>>> e=jlist.add('test')
>>> e=jlist.add(1)
>>> jlist.add(0,'new test')
>>> e=jlist.add(2)
>>> print jlist
[new test, test, 1, 2]
>>> print jlist.get(0)
new test
>>> print jlist.toArray()
array(java.lang.Object, ['new test', 'test', 1, 2])
```

You may notice that when we append an element to the end of this list, we assign the result to the variable `'e'`. In Jython, it returns `'1'` for success (or `true` for Java). We also can add an object `obj` at the position characterized with the index `i` using the method `add(i, obj)`. Analogously, one can access elements by their integer positions. For example, one can retrieve an object back using the method `get(i)`. The list of elements can be retrieved in a loop exactly as we usually do for the Jython lists. Let us show a more complete example below:

#### Java list example

```
from java.util import *

jlist=ArrayList()
# append integers
for i in range(100):
    jlist.add(i)
print jlist.size()

# replace at 0 position
jlist.set(0,100)
s=jlist
print type(s)
```

```
# range between 0-50
newlist=jlist.subList(0,50)
for j in newList:
    print j
```

Run the above code and make sense of its output.

Probably, there are not too strong reasons to use `Java List` while working with Jython, since the native Jython list discussed in the previous section should be sufficient for almost any task. However, it is possible that you will need to use Java lists in order to integrate your application natively into the Java platform after moving your code into a pure Java coding.

### 2.7.1.1 Sorting Java Lists

One can do several manipulations with the `List` using the `Java Collection` class. Below we show how to sort a list using the natural ordering of its elements, and how to reverse the order:

```
>>> from java.util import *
>>> jlist=ArrayList()
>>> jlist.add('zero'); jlist.add('one'); jlist.add('two')
>>> Collections.sort(jlist)
>>> print jlist
>>> [one, two, zero]
>>> Collections.reverse(jlist)
>>> print jlist
>>> [zero, two, one]
```

The next question is how to sort a list with more complicated objects, using some object attribute for sorting. Consider a list containing a sequence of other lists as in the case shown below:

```
>>> from java.util import *
>>> jlist=ArrayList()
>>> jlist.add([2,2]); jlist.add([3,4]); jlist.add([1,1])
>>> print jlist
[[2, 2], [3, 4], [1, 1]]
```

Here there is a small problem: how can we tell to the method `sort()` that we want to perform a sorting using a first (or second) item in each element-list? Or, more generally, if each element is an instance of some class, how can we change ordering objects instantiated by the same class?

One can do this by creating a small class which implements the `Comparator` interface. We will consider Jython classes in Sect. 2.11, so at this moment just accept

this construction as a simple prescription that performs a comparison of two objects. The method `compare(obj1, obj2)` of this class compares objects and returns a negative value, zero, or a positive integer value depending on whether the object is less than, equal to, or greater than the specified object. Of course, it is up to you to define how to perform such object comparison. For the example above, each object is a list with two integers, so one can easily prototype a function for object comparison. Let us write a script which orders the list in increasing order using the first element of each list:

Sorting Java lists

```
from java.util import *

jlist=ArrayList()
jlist.add([2,2]); jlist.add([3,4]); jlist.add([1,1])

class cmt(Comparator):
    def compare(self, i1,i2):
        if i1[0]>i2[0]: return 1
        return 0

Collections.sort(jlist,cmt())
print jlist
```

After running this script, all elements will be ordered and the print method displays `[[1, 1], [2, 2], [3, 4]]`.

We will leave the reader here. One can always find further information about the Java lists from any Java textbook.

### 2.7.2 Set. A Collection Without Duplicate Elements

The Set container from the package `java.util` is a Java collection that cannot contain duplicate elements. Such set can be created using general-purpose implementations based on the `HashSet` class:

```
>>> from java.util import *
>>> s=HashSet()
>>> e=s.add('test')
>>> e=s.add('test')
>>> e=s.add(1)
>>> e=s.add(2)
>>> print s
[1, 2, test]
```

As you can see from this example, the string `'test'` is automatically removed from the collection. Operations with the Java sets are exactly the same as those with

the `ArrayList`. One can loop over all elements of the set collection using the same method as that used for the `ArrayList` class, or one can use a method by calling each element by its index:

```
>>> for i in range(s.size()):
>>> ...print s[i]
```

As in the case with the Java lists, you may face a problem when go beyond simple items in the collection. If you want to store complicated objects with certain attributes, what method should be used to remove duplicates? You can do this as well but make sure that instances of the class used as elements inside the Java set use hash tables (most of them do). In case of the example shown in Sect. 2.7.1.1, you cannot use `HashSet` since lists are unhashable. But with tuples, it is different: Tuples have hash tables, so the code snippet below should be healthy:

```
>>> from java.util import *
>>> s=HashSet()
>>> e=s.add( (1,2) )
>>> e=s.add( (2,4) )
>>> e=s.add( (1,2) )
>>> print s
[(2, 4), (1, 2)]
```

As you can see, the duplicate entry `(1, 2)` is gone from the container. In case if you need to do the same with Python lists, convert them first into tuples as shown in Sect. 2.6.5.

### 2.7.3 *SortedSet. Sorted Unique Elements*

Next, why not to keep all our elements in the Java set container in a sorted order, without calling an additional sorting method each time we add a new element? The example below shows the use of the `SortedSet` Java class:

```
>>> from java.util import *
>>> s=TreeSet()
>>> e=s.add(1)
>>> e=s.add(4)
>>> e=s.add(4)
>>> e=s.add(2)
>>> print s
[1, 2, 4]
```

the second value “4” is automatically removed and the collection appears in the sorted order.

### 2.7.4 Map. Mapping Keys to Values

As it is clear from the title, now we will consider the Java Map collection which maps keys to specific objects. This collection is analogous to a Jython dictionary, Sect. 2.6.7. Thus, a map cannot contain duplicate keys as we have learned from the Jython dictionaries.

Let us build a map collection based on the HashMap Java class:

```
>>> from java.util import *
>>> m=HashMap()
>>> m.put('a', 1)
>>> m.put('b', 2)
>>> m.put('c', 3)
>>> print m
{b=2, c=3, a=1}
```

Now you can see that Java maps have the same functionality as the Jython dictionaries. As for any Java collection, the size of the Map is given by the method `size()`. One can access the map values using the key:

```
>>> print m['a']
1
```

Similar to the lists, one can print all keys in a loop:

```
>>> for key in m:
>>> ... print key, 'corresponds to', m[key]
b corresponds to 2
c corresponds to 3
a corresponds to 1
```

Here we print all keys and also values corresponding to the keys.

### 2.7.5 Java Map with Sorted Elements

This time we are interested in a map with sorted keys. For this one should use the class `TreeMap` and the same methods as for the `HashMap` class discussed before:

```
>>> from java.util import *
>>> m=TreeMap()
>>> m.put('c', 1)
>>> m.put('a', 2)
```

```
>>> m.put('b', 3)
>>> print m
{a=2, b=3, c=1}
```

Compare this result with that given in the previous subsection. Now the map is sorted using the keys.

### 2.7.6 Real Life Example: Sorting and Removing Duplicates

Based on the Java methods discussed above, we can do something more complicated. In many cases, we need to deal with a sequence of data records. Each record, or event, can consist of strings, integer and real numbers. So we are dealing with lists of lists. For example, assume we record one event and make measurements of this event by recording a string describing some feature and several numbers characterizing this feature. Such example was already considered in Sect. 2.6.4.1.

Assume we make many such observations. What we want to do at the end of our experiment is to remove duplicates based on the string with a description, and then sort all the records (or observations) based on this description. This looks like a real project, but not for Jython! The code below does everything using a several lines of the code:

— Sorting and removing duplicates —

```
from java.util import *

data=ArrayList()
data.add( ["star",1.1,30] )
data.add( ["galaxy",2.2,80] )
data.add( ["galaxy",3.3,10] )
data.add( ["moon",4.4,50] )

map=TreeMap()
for row in data:
    map.put(row[0],row[1:])

data.clear()
for i in map:
    row=map[i]
    row.insert(0,i)
    data.add(row)

print data
```

Let us give some explanations. First, we make a data record based on the list 'data' holding all our measurements. Then we build a `TreeMap` class and use the first element to keep the description of our measurement in form of a "key".

The rest of our record is used to fill the map values (see `row[1:]`). As you already know, when we fill the `TreeMap` object, we remove duplicate elements and sort the keys automatically. Once the map is ready, we remove all entries from the list and refill it using a loop over all the keys (which are now ordered). Then we combine the key value to form a complete event record. The output of the script is given below:

```
[['galaxy', 3.3, 10], ['moon', 4.4, 50], ['star', 1.1, 30]]
```

We do not have extra record with the description 'galaxy' and, expectedly, all our records are appropriately sorted.

## 2.8 Random Numbers

A generation of random numbers is an essential phase in scientific programming. Random numbers are used for estimating integrals, generating data encryption keys, data interpretation, simulation and modeling complex phenomena. In many examples of this book, we will simulate random data sets for illustrating data-analysis techniques.

Let us give a simple example which shows how to generate a random floating point number in the range  $[0, 1]$  using the Python language:

```
>>> from random import *
>>> r=Random()
>>> r.randint(1,10) # a random number in range [0.10]
```

Since we do not specify any argument for the `Random()` statement, a random seed from the current system time is used. In this case, every time you execute this script, a new random number will be generated.

In order to generate a random number predictably for debugging purpose, one should pass an integer (or long) value to an instance of the `Random()` class. For the above code, this may look as: `r=Random(100L)`. Now the behavior of the script above will be different: every time when you execute this script, the method `randint(1,10)` will return the same random value, since the seed value is fixed.

Random numbers in Python can be generated using various distributions depending on the applied method:

```
>>> r.random()           # in range [0.0, 1.0)
>>> r.randint(min,max)   # int in range [min,max]
>>> r.uniform(min,max)   # real number in [min,max]
>>> r.betavariate(a,b)    # Beta distribution (a>0,b>0)
>>> r.expovariate(lambda) # Exponential distribution
>>> r.gauss(m,s)          # Gaussian distribution
>>> r.lognormvariate(m,s) # Log normal distribution
```

```
>>> r.normalvariate(m,s) # Normal distribution
>>> r.gammavariate(a, b) # Gamma distribution.
>>> r.seed(i)           # set seed (i integer or long)
>>> state=r.getstate()  # returns internal state
>>> setstate(state)     # restores internal state
```

In the examples above, 'm' denotes a mean value and 's' represents a standard deviation for the output distributions.

Random numbers are also used for manipulations with Jython lists. One can randomly rearrange elements in a list as:

```
>>> list=[1,2,3,4,5,6,7,8,9]
>>> r.shuffle(list)
>>> print list
[3, 4, 2, 7, 6, 5, 9, 8, 1] # random list
```

One can pick up a random value from a list as:

```
>>> list=[1,2,3,4,5,6,7,8,9]
>>> r.choice(list) # get a random element
```

Similarly, one can get a random sample of elements as:

```
>>> list=[1,2,3,4,5,6,7,8,9]
>>> print r.sample(list,4) # random list
>>> [4, 2, 3, 6]
```

Of course, the printed numbers will be different in your case.

## 2.9 Time Module

The time module is rather popular due to several reasons. First, it is always a good idea to find out the current time. Secondly, it is an essential module for more serious tasks, such as optimization and benchmarking analysis programs or their parts. Let us check the methods of the module time:

```
>>> import time
>>> dir(time) # check what is inside
['__doc__', 'accept2dyear', 'altzone', 'asctime',
'classDictInit', 'clock', 'ctime', 'daylight',
'gmtime', 'locale_asctime', 'localtime', 'mktime',
'sleep', 'strftime', 'struct_time', 'time',
'timezone', 'tzname']
```

You may notice that there is a method called `__doc__`. This looks like a method to keep the documentation for this module. Indeed, by printing the documentation of this module as

```
>>> print time.__doc__
```

you will see a rather comprehensive description. Let us give several examples:

```
>>> time.time() # time in seconds since the Epoch
>>> time.sleep() # delay for a number of seconds
>>> t=time.time()
>>> print t.strftime('4-digit year: %Y, 2-digit year: \
                        %y, month: %m, day: %d')
```

The last line prints the current year, the month and the day with explanatory annotations.

To find the current day, the easiest is to use the module `datetime`:

```
>>> import datetime
>>> print "The date is", datetime.date.today()
>>> The date is 2008-11-14
>>> t=datetime.date.today()
>>> print t.strftime("4-digit year: \
                        %Y, 2-digit year: %y, month: %m, day: %d")
```

To force a program to sleep a certain number of seconds, use the `sleep()` method:

```
>>> seconds = 10
>>> time.sleep(seconds)
```

### 2.9.1 Benchmarking

For tests involving benchmarking, i.e. when one needs to determine the time spent by a program or its part on some computational task, one should use a Jython module returning high-resolution time. The best is to use the module `clock()` which returns the current processor time as a floating point number expressed in seconds. The resolution is rather dependent on the platform used to run this program but, for our benchmarking tests, this is not too important.

To benchmark a piece of code, enclose it between two `time.clock()` statements as in this code example:

```
>>> start = time.clock(); \
    [SOME CODE FOR BENCHMARKING]; \
    end = time.clock()
>>> print 'The execution of took (sec) =', end-start
```

Let us give a concrete example: We will benchmark the creation of a list with integer numbers. For benchmarking in an interactive mode, we will use the `exec()` statement. This code benchmarks the creation of a list with the integer numbers from 0 to 99999.

```
>>> code='range(0,100000)'
>>> start=time.clock();List=exec(code);end=time.clock()
>>> print 'Execution of the code took (sec)=',end-start
Execution of the code took (sec) = 0.003
```

Alternatively, one can write this as:

```
>>> List=[]
>>> code='for x in range(0,100000): List.append(x)'
>>> start=time.clock();exec(code);end=time.clock()
>>> print 'Execution of the code took (sec)=',end-start
```

## 2.10 Python Functions and Modules

Jython supports code reuse via functions and classes. The language has many built-in functions which can be used without calling the `import` statement. For example, the function `dir()` is a typical built-in function. But how one can find out which functions have already been defined? The `dir()` itself cannot display them. However, one can always use the statement `dir(module)` to get more information about a particular module. Try to use the lines:

```
>>> import __builtin__
>>> dir(__builtin__)
...'compile', 'dict', 'dir', 'eval' ..
```

This prints a rather long list of the built-in functions available for immediate use (we show here only a few functions).

Other (“library”) functions should be explicitly imported using the `import` statement. For example, the function `sqrt()` is located inside the package ‘`math`’, thus it should be imported as ‘`import math`’. One can always list

all functions of a particular package by using the `dir()` function as shown in Sect. 2.2.3.

It is always a good idea to split your code down into a series of functions, each of which would perform a single logical action. The functions in Jython are declared using the statement `def`. Here is a typical example of a function which returns  $(a-b)/(a+b)$ :

```
>>>def func(a,b):
>>> ... "function"
>>> ... d=(a-b)/(a+b)
>>> ... return d
>>>print func(3.0,1.0)
0.5
>>> print func.__doc__
function
```

To call the function `func()`, a comma-separated list of argument values is used. The 'return' statement inside the function definition returns the calculated value back and exits the function block. If no return statement is specified, then 'None' will be returned. The above function definition contains a string comment 'function'. A function comment should always be on the first line after the `def` attributed. One can print the documentation comment with the method `__doc__` from a program from which the function is called.

One can also return multiple values from a function. In this case, put a list of values separated by commas; then a function returns a tuple with values as in this example:

```
>>>def func(a,b,c=10):
>>> ... d1=(a-b)/(a+b)
>>> ... d2=(a*b*c)
>>> ... return d1,d2
>>>print func(2,1)
(0, 20)
>>> >print func(2.,1.0)
(0.5, 30.0)
```

The example shows another features of Jython functions: the answer from the function totally depends on the type of passed argument values. The statement `func(2,1)` interprets the arguments as integer values, thus the answer for  $(a-b)/(a+b)$  is zero (not the expected 0.5 as in case of double values). Thus, Jython functions are *generic* and any type can be passed in.

One can note another feature of the above example: it is possible to omit a parameter and use default values specified in the function definition. For the above example, we could skip the third argument in the calling statement, assuming `c=10` by default.

All variable names assigned to a function are local to that function and exist only inside the function block. However, you may use the declaration 'global' to force a variable to be common to all functions.

```
>>>def func1(a,b):
>>> ... global c
>>> ... return a+b+c
>>>def func2(a,b):
>>> ... global c
>>> ... c=a+b
>>>
>>>print func2(2,1)
None
>>>print func1(2,1)
6
```

Thus, once the global variable 'c' is assigned a value, this value is propagated to other functions in which the 'global' statement was included. The second function does not have the 'return' statement, thus it returns 'None'.

We should note that a function in Jython can call other functions, including itself.

In Jython, one can also create an anonymous function at runtime, using a construct called 'lambda' discussed in Sect. 2.6.6. The example below shows two function declarations with the same functionality. In one case, we define the function using the standard ("named") approach, and the second case uses the "lambda" anonymous declaration:

```
>>> def f1 (x): return x*x
>>> print f1(2)
4
>>> f1=lambda x: x*x
>>> print f1(2)
4
```

Both function definitions, f1 and f2 do exactly the same operation. However, the "lambda" definition is shorter.

It is very convenient to put functions in files and use them later in your programs. A file containing functions (or any Jython statement!) should have the extension '.py'. Usually, such file is called a "module". For example, one can create a file 'Func.py' and put these lines:

```
File 'Func.py'

def func1(a,b):
    "My function 1"
    global c
    return a+b+c

def func2(a,b):
```

```
"My function 2"  
global c  
c=a+b
```

This module can be imported into other modules. Let us call this module from the JythonShell prompt with the following commands:

```
>>> import Func  
>>> print Func.func2(2,1)  
None  
>>> print Func.func1(2,1)  
6
```

We can access functions exactly as if they are defined in the same program, since the `import` statement executes the file `'Func.py'` and makes the functions available at runtime of your program.

Probably, we should remind again that one can import all functions with the statement `'from Func import *'`, as we usually do in many examples of this book. In this case, one can call the functions directly without typing the module name.

Another question is where such modules should be located? How can we tell Jython to look at particular locations with module files? This can be done by using a predefined list `sys.path` from the `'sys'` module. The list `sys.path` contains strings that specify the location of Jython modules. One can add an additional module location using the `append()` method: In this example, we added the location  `'/home/lib'` and printed out all directories containing Jython modules:

```
>>> import sys  
>>> sys.path.append('/home/lib')  
>>> print sys.path
```

Here we have assumed that we put new functions in the directory  `'/home/lib'`.

Now we are equipped to go further. We would recommend to read any book about Python or Jython to find more detail about Jython modules and functions.

## 2.11 Python Classes

As for any object-oriented language, one can define a Jython class either inside a module file or inside the body of a program. Moreover, one can define many classes inside a single module.

Classes are templates for creation of objects. Class attributes can be hidden, so one can access the class itself only through the methods of the class. Any Python book or tutorial should be fine in helping to go into the depth of this subject.

A Jython class is defined as:

```
>>> class ClassName[args]:  
>>> ... [code block]
```

where [code block] indicates the class body and bounds its variables and methods.

The example below shows how to create a simple class and how to instantiate it:

```
>>> class Func:  
>>> ... 'My first class'  
>>> ... a='hello'; b=10  
>>>  
>>> c=Func()  
>>> print c.a, c.b  
hello 10
```

The class defined above has two public variables, `a` and `b`. We create an instance of the class 'Func' and print its public attributes, which are just variables of the type string and integer. As you can see, the class instance has its own namespace which is accessible with the dot. As for functions and modules, classes can (and should) have documentary strings.

The created instance has more attributes which can be shown as a list using the built-in function `dir()`: Try this line:

```
>>> dir(Func)  
['__doc__', '__module__', 'a', 'b']
```

The command displays the class attributes and the attributes of its class base. Note that one can also call the method `dir(obj)`, where 'obj' is an instance of the class (`c` in our example), rather than explicitly using the class name.

But what about the attributes which start from the two leading underscores? In the example above, both variables, `a` and `b`, are public, so they can be seen by a program that instantiates this class. In many cases, one should have private variables seen by only the class itself. For this, Jython has a naming convention: one can declare names in the form `__Name` (with the two leading underscores). Such convention offers only the so-called "name-mangling" which helps to discourage internal variables or methods from being called from outside a class.

In the example above, the methods with two leading underscores are private attributes generated automatically by Jython during class creation. The variable `__doc__` keeps the comment line which was put right after the class definition,

and the second variable `__module__` keeps a reference to the module where the class is defined.

```
>>> print c.__doc__
My first class
>>> print c.__module__
None
```

The last call returns 'None' since we did not put the class in an external module file.

### 2.11.1 *Initializing a Class*

The initialization of a Jython class can be done with the `__init__` method, which takes any number of arguments. The function for initialization is called immediately after creation of the instance:

```
>>> class Func():
>>>     'My class with initialization'
>>>     def __init__(self, filename=None):
>>>         self.filename=filename
>>>     def __del__(self):
>>>         # some close statement goes here
>>>     def close(self):
>>>         # statement to release some resources
```

Let us take a closer look at this example. You may notice that the first argument of the `__init__` call is named as `self`. You should remember this convention: every class method, including `__init__`, is always a reference to the current instance of the class.

In case if an instance was initialized and the associated resources are allocated, make sure they are released at the end of a program. This is usually done with the `__del__` method which is called before Jython garbage collector deallocates the object. This method takes exactly one parameter, `self`. It is also a good practice to have a direct cleanup method, like `close()` shown in this example. This method can be used, for example, to close a file or a database. It should be called directly from a program which creates the object. In some cases, you may wish to call `close()` from the `__del__` function, to make sure that a file or database was closed correctly before the object is deallocated.

### 2.11.2 *Classes Inherited from Other Classes*

In many cases, classes can be inherited from other classes. For instance, if you have already created a class 'exam1' located in the file 'exam1.py', you can use this class to build a new ("derived") class as:

```
>>> from exam1 import exam1
>>> class exam2(exam1):
>>> ... [class body]
```

As you can see, first we import the ancestor class 'exam1', and then the ancestor of the class is listed in parentheses immediately after the class name. The new class 'exam2' inherits all attributes from the 'exam1' class. One can change the behavior of the class 'exam1' by simply adding new components to 'exam2' rather than rewriting the existing ancestor class. In particular, one can overwrite methods of the class 'exam1' or even add new methods.

### 2.11.3 *Java Classes in Jython*

The power of Jython, a Java implementation of the Python language, becomes clear when we start to call Java classes using Python syntax. Jython was designed as a language which can create instances of Java classes and has an access to any method of such Java class.

This is exactly what we are going to do while working with the jHepWork libraries. The example below shows how to create the Java Date object from java.util and use its methods:

```
>>> from java.util import Date
>>> date=Date()
>>> date.toGMTString()
'09 Jun 2009 03:48:17 GMT'
```

One can use the code assist to learn more about the methods of this Java class (Type the object name followed by a dot and use [Ctrl]+[Space] in JythonShell for help. Similarly, one can call dir(obj), where obj is an object which belongs to the Java platform. For jHepWork IDE code editor, use a dot and the key [F4].

In this book, we will use Java-based numerical libraries from jHepWork, thus most of the time we will call Java classes of this package. Also, in many cases, we call classes from the native Java platform. For example, the AWT classes 'Font' and 'Color' are used by many jHepWork objects to set fonts and colors. For example, Sect. 3.3.1 shows how to build a Java instance of graphical canvas based on the Java class HPlot.

### 2.11.4 Topics Not Covered

In this book, we will try to avoid going into the depths of Python classes. We cannot cover here many important topics, such as inheritance (the ability of a class to inherit properties from another class) and abstract classes. We would recommend any Python or Jython textbook to learn more about classes.

As we have mentioned before, we would recommend to develop Java libraries to be linked with Jython, rather than building numerical libraries using pure-Jython classes; for the latter approach, you will be somewhat locked inside the Python language specification, plus this may result in slow overall performance of your application. Of course, you have to be familiar with the Java language in order to develop Java classes.

## 2.12 Used Memory

To know how much memory used by the Java virtual machine for an application is important for code debugging and optimization. The amount of memory currently allocated to a process can be found using the standard Java library as in the example below:

```
>>> from java.lang import Runtime
>>> r=Runtime.getRuntime()
>>> Used_memory = r.totalMemory() - r.freeMemory()
>>> 'Used memory in MB = ', Used_memory/(1024*1024)
```

We will emphasize that this only can be done in Jython, but not in CPython which does not have any knowledge about the Java virtual machine.

We remind that if you use the jHepWork IDE, one can look at the memory monitor located below the code editor.

## 2.13 Parallel Computing and Threads

A Jython program can perform several tasks at once using the so-called threads. A thread allows to make programs parallelizable, thus one can significantly boost their performance using parallel computing on multi-core processors.

Jython provides a very effective threading compared to CPython, since JAVA platform is designed from the ground up to support multi-thread programming. A multi-threading program has significant advantage in processing large data sets, since one can break up a single task into pieces that can be executed in parallel. At the end, one can combine the outputs. We will consider one such example in Sect. 16.4.

To start a thread, one should import the Jython module 'threading'. Typically, one should write a small class to create a thread or threads. The class should contain the code to be executed when the thread is called. One can also put an initialization method for the class to pass necessary arguments. In the example below, we create ten independent threads using Jython. Each thread prints integer numbers. We create instances of the class shown above and start the thread using the method `start()` which executes the method `run()` of this class.

\_\_\_\_\_ A thread example \_\_\_\_\_

```
from threading import Thread

class test(Thread):
    def __init__(self,fin):
        Thread.__init__(self)
        self.fin = fin
    def run(self):
        print 'This is thread No='+str (self.fin)

for x in xrange ( 10 ):
    current=test(x)
    current.start()
print 'done!'
```

Here we prefer to avoid going into detailed discussion of this topic. Instead, we will illustrate the effectiveness of multi-threading programs in the following chapters when we will discuss concrete data-analysis examples.

## 2.14 Arrays in Jython

This is an important section: Here we will give the basics of objects which can be used for effective numerical calculations and storing consecutive values of the same type.

Unfortunately, the Java containers discussed in Sect. 2.7 cannot be used in all cases. Although they do provide a handy interface for passing arrays to Java and jHepWork objects to be discussed later, they do not have sufficient number of built-in methods for manipulations.

Jython lists can also be used for data storage and manipulation. However, they are best suited for general-purpose tasks, such as storing complex objects, especially if they belong to different types. They are rather heavy and slow for numerical manipulations with numbers.

Here we will discuss Python/Jython arrays that can be used for storing a sequence of values of a certain type, such as integers, long values, floating point numbers etc. Unlike lists, arrays cannot contain objects with different types.

The Jython arrays directly mapped to Java arrays. If you have a Java function which returns an array of double values, and declared as `double[]` in a Java code, this array will be seen by Jython as an array.

**Table 2.3** Characters used to specify types of the Jython arrays

Jython array types	
Typecode	Java type
z	boolean
c	char
b	byte
h	short
i	int
l	long
f	float
d	double

To start working with the arrays, one should import the module `jarray`. Then, for example, an array with integers can be created as:

```
>>> from jarray import *
>>> a=array([1,2,3,4], 'i')
```

This array, initialized from the input list `[1, 2, 3, 4]`, keeps integer values, see the input character `'i'` (integer). To create an array with double values, the character `'i'` should be replaced by `'d'`. Table 2.3 shows different choices for array types. The length of arrays is given by the method `len(a)`.

Arrays can be initialized without invoking the lists. To create an array containing, say, ten zeros, one can use this statement:

```
>>> a=zeros(10, 'i')
```

here, the first argument represents the length of the array, while the second specifies its type.

A new value `'val'` can be appended to the end of an array using the `append(val)` method if the value has exactly the same type as that used during array creation. A value can be inserted at a particular location given by the index `'i'` by calling the method `insert(i, val)`. One can also append a list to the array by calling the method `fromlist(list)`.

The number of occurrences of a particular value `'val'` in an array can be given by the method `count(val)`. To remove the first occurrence of `'val'` from an array, use the `remove(val)` method.

### 2.14.1 Array Conversion and Transformations

Many Java methods return Java arrays. Such arrays are converted to Jython arrays when Java classes are called from a Jython script.

Very often, it is useful to convert arrays to Jython list for easy manipulation. Use the method `tolist()` as below:

```
>>> from jarray import *
>>> a=array([1,2,3,4], 'i')
>>> print a.tolist()
[1, 2, 3, 4]
```

One can reverse all elements in arrays using the `reverse()` method. Finally, one can also transform an array into a string applying the `toString()` method.

There are no too many transformations for Jython arrays: in the following chapters, we will consider another high-level objects which are rather similar to the Jython arrays but have a large number of methods for numerical calculations.

### 2.14.2 Performance Issues

We have already noted that in order to achieve the best possible performance for numerical calculations, one should use the built-in methods, rather than Python-language constructs.

Below we show a simple benchmarking test in which we fill arrays with one million elements. We will consider two scenarios: In one case, we use a built-in function. In the second case, we use a Python-type loop. The benchmarking test was done using the `time` module discussed in Sect. 2.9. The only new component in this program is the one in which we format the output number: here we print only four digits after the decimal point.

#### Benchmarking Jython arrays

```
import time
from jarray import *

start=time.clock()
a=zeros(1000000, 'i')
t=time.clock()-start
print 'Build-in method (sec)= %.4f' % t

start=time.clock()
a=array([], 'i')
for i in range(0,1000000,1):
    a.append(0)
```

```
t=time.clock()-start
print 'Python loop (sec) %.4f' % t
```

Run this small script by loading it in the editor and using the “[run]” button. The performance of the second part, in which integers are sequentially appended to the array, is several orders of magnitudes slower than for the case with the built-in array constructor `zeros()`.

Generally, the performance of Jython loops is not so dramatically slow. For most examples to be discussed later, loops are several times slower than equivalent loops implemented in built-in functions.

## 2.15 Exceptions in Python

Exception is an unexpected error during program execution. An exception is raised whenever an error occurs.

Jython handles the exceptions using the “try”-“except”-“else” block. Let us give a short example:

```
>>> b=0
>>> try:
>>> ... a=100/b
>>> except:
>>> ... print "b is zero!"
```

Normally, if you will not enclose the expression `a=100/b` in the “try”-“except” block, you will see the message such as:

```
>>> a=100/b
Traceback (innermost last):
  File "<input>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
```

As you can see, the exception in this case is `ZeroDivisionError`.

Another example of the exceptions is “a file not found” which happens while attempting to open a non-existing file (see the next chapter describing Jython I/O). Such exception can be caught in a similar way:

```
>>> try:
>>> ... f=open('filename')
>>> except IOError, e:
>>> ... print e
```

This time the exception is `IOError`, which was explicitly specified. The variable `e` contains the description of the error.

Exceptions can be rather different. For example, `NameError` means unknown name of the class or a function, `TypeError` means operation for incompatible types and so on. One can find more details about the exceptions in any Python manual.

## 2.16 Input and Output

### 2.16.1 User Interaction

A useful feature you should consider for your Jython program is interactivity, i.e. when a program asks questions at run-time and a user can enter desired values or strings. To pass a value, use the Jython method `input()`:

```
>>> a=input('Please type a number: ')
>>> print 'Entered number=',a
```

In this example, the `input()` method prints the string 'Please type a number:' and waits for the user response.

But what if the entered value is not a number? In this case, we should handle an exception as discussed in Sect. 2.15.

If you want to pass a string, use the method `raw_input()` instead of `input()`.

The above code example works only for the stand-alone Jython interpreter, outside the jHepWork IDE. For the jHepWork IDE, this functionality is not supported. In fact, you do not need this feature at all: When working with the IDE, you are already working in an interactive mode. However, when you run Jython using the system prompt, the operations `input()` or `raw_input()` are certainly very useful.

### 2.16.2 Reading and Writing Files

File handling in Jython is relatively simple. One can open a file for read or write using the `open()` statement:

```
>>> f=open(fileName, option)
```

where 'FileName' represents a file name including the correct path, 'option' is a string which could be either 'w' (open for writing, old file will be removed), 'r' (open for reading) or 'a' (file is opened for appending, i.e. data written to it is added on at the end). The file can be closed with the `close()` statement.

Let us read a file 'data.txt' with several numbers, each number is positioned on a new line:

```
>>> f=open('data.txt','r')
>>> s=f.readline()
>>> x1=float(s)
>>> s=f.readline()
>>> x2=float(s)
>>> f.close()
```

At each step, `readline()` reads a new line and returns a string with the number, which is converted into either a float or integer.

The situation is different if several numbers are located on one line. In the simple case, they can be separated by a space. For such file format, we should split the line into pieces after reading it. For example, if we have two numbers separated by white spaces in one line, like '100 200', we can read this line and then split it as:

```
>>> f=open('data.txt','r')
>>> s=f.readline()
>>> x=s.split()
>>> print s
['100','200']
```

As you can see, the variable 'x' is a list which contains the numbers in form of strings. Next, you will need to convert the elements of this list into either float or integer numbers:

```
>>> x1=float( x[0] )
>>> x2=float( x[1] )
```

In fact, the numbers can also be separated by any string, not necessarily by white spaces. Generally, use the method `split(str)`, where 'str' is a string used to split the original string.

There is another powerful method: `readlines()`, which reads all lines of a file and returns them as a list:

```
>>> f=open('data.txt')
>>> for l in f.readlines():
>>> ... print l
```

To write numbers or strings, use the method `write()`. Numbers should be coerced into strings using the `str()` method. Look at the example below:

```
>>> f=open('data.txt','w')
>>> f.write( str(100)+'\n' )
```

```
>>> f.write( str(200) )
>>> f.close()
```

here we added a new line symbol, so the next number will be printed on a new line.

One can also use the statement 'print' to redirect the output into a file. This can be done with the help of the >> operator. (Note: by default, this operator prints to a console.). Let us give one example that shows how to print ten numbers from zero to nine:

```
>>> f=open('data.txt', 'w')
>>> for i in range(10):
>>> ...     print >> f, i
>>> f.close()
```

One can check the existence of the file using the Jython module 'os':

```
>>> import os
>>> b=os.path.exists(fileName)
```

where 'b=0' (false in Java) if the file does not exist, and 'b=1' (true in Java) in the opposite case.

### 2.16.3 Input and Output for Arrays

Jython arrays considered in the previous section can be written into an external (binary) file. Once written, one can read its content back to a new array (or append the values to the existing array).

```
>>> from jarray import *
>>> a=array([1,2,3,4], 'i')
>>> f=open('data.txt', 'w')
>>> a.tofile(f)          # write values to a file
>>> f.close()
>>> # read values
>>> f=open('data.txt', 'r')
>>> b=array([], 'i')
>>> b.fromfile(f,3) # read 3 values from the file
>>> print b
array('i', [1, 2, 3])
```

It should be noted that the method fromfile() takes two arguments: the file object and the number of items (as machine values).

### 2.16.4 Working with CSV Python Module

The CSV (“Comma Separated Value”) file format is often used to store data structured in a table. It is used for import and export in spreadsheets and databases and to exchange data between different applications. Data in such files are either separated by commas, tabs, or some custom delimiters.

Let us write a table consisting of several rows. We will import Python `csv` file and write several lists with values using the code below:

Writing a CSV file

```
import csv

w=csv.writer(open('test.csv', 'w'),delimiter=',')
w.writerow(['London', 'Moscow', 'Hamburg'])
w.writerow([1,2,3])
w.writerow([10,20,30])
```

Execute this script and look at the current directory. You will see the file `'test.csv'` with the lines:

```
London,Moscow,Hamburg
1,2,3
10,20,30
```

This is expected output: each file entry is separated by a comma as given in the `delimiter` attribute specified in our script. One can put any symbol as a delimiter to separate values. The most popular delimiter is a space, tab, semi-column and the symbol `'|'`. The module also works for quoted values and line endings, so you can write files that contain arbitrary strings (including strings that contain commas). For example, one can specify the attribute `quotechar='|'` to separate fields containing quotes.

In the example below we read a CSV file and, in case of problems, we print an error message using the Python exception mechanism discussed in Sect. 2.15:

Reading a CSV file

```
import csv

r = csv.reader(open('test.csv', 'rb'), delimiter=',')
try:
    for row in r:
        print row
except csv.Error, e:
    print 'line %d: %s' % (reader.line_num,e)
```

Let us convert our example into a different format. This time we will use a double quote (useful when a string contains comma inside!) for each value and tab for value

separations. The conversion script is based on the same Jython `csv` module and will look as:

```

Converting CSV file
import csv

reader=csv.reader(open('test.csv','rb'),delimiter=',')
writer=csv.writer(open('newtest.csv','wb'),\
                    delimiter='\t',\
                    quotechar='"', quoting=csv.QUOTE_ALL)

for row in reader:
    writer.writerow(row)

```

The output file will look as:

```

"London"  "Moscow"  "Hamburg"
"1"       "2"       "3"
"10"      "20"      "30"

```

But what if we do not know which format was used for the file you want to read in? First of all, one can always open this file in an editor to see how it looks like, since the CSV files are human readable. One can use the `jHepWork` editor by printing this line in the JythonShell prompt:

```

>>> view.open('newtest.csv', 0 )

```

which opens the file `'newtest.csv'` in the IDE. Alternatively, one can determine the file format automatically using the `Sniffer` method for safe opening of any CSV file:

```

Reading a CSV file using sniffer
import csv

f=open('newtest.csv')
dialect = csv.Sniffer().sniff(f.read(1024))
f.seek(0)
reader = csv.reader(f, dialect)
for row in csv.reader(f, dialect):
    print row

```

This time we do not use the exception mechanism, since it is very likely that your file will be correctly processed.

We will come back to the CSV file format in the following chapters when we will discuss Java libraries designed to read the CSV files.

### 2.16.5 Saving Objects in a Serialized File

If you are dealing with an object from the Python-language specification, you may want to store this object in a file persistently (i.e. permanently), so another application can read it later. In Jython, one can serialize (or pickle) an object as:

```
>>> import pickle
>>> f=open('data.pic','w')
>>> a=[1,2,3]
>>> pickle.dump(a,f)
>>> f.close()
```

One can restore the object back as:

```
>>> import pickle
>>> f=open('data.pic','r')
>>> a=pickle.load(f)
>>> f.close()
```

In this example, we save a list and then restore it back from the file 'data.pic'. One cannot save Java objects using the same approach. Also, any object which has a reference to a Java class cannot be saved. We will consider how to deal with such special situations in the following chapters.

### 2.16.6 Storing Multiple Objects

To store one object per file is not too useful feature. In many cases, we are dealing with multiple objects. Multiple objects can be stored in one serialized file using the `shelve` module. This Jython module can be used to store anything that the pickle module can handle.

Let us give one example in which we store two Jython objects, a string and a list:

```
>>> import shelve
>>> sh=shelve.open('data.shelf')
>>> sh['describe']='My data'
>>> sh['data']=[1,2,3,4]
>>> sh.close()
```

The example above creates two files, 'data.shelf.dir' and 'data.shelf.dat'. The first file contains a "directory" with the persistent data. This file is in a human-readable form, so if you want to learn what is stored inside of the data file, one can open it and read its keys. For the above example, the file contains the following lines:

```
'describe', (0, 15)
'data', (512, 22)
```

The second file, 'data.shelf.dat', contains the actual data in a binary form.

One can add new objects to the “shelf” file. In the example below, we add a Jython map to the existing file:

```
>>> import shelve
>>> sh=shelve.open('data.shelf')
>>> sh['map']={'x1':100,'x2':200}
>>> sh.close()
```

Let us retrieve the information from the shelve storage and print out all saved objects:

```
>>> import shelve
>>> sh=shelve.open('data.shelf')
>>> for i in sh.keys():
>>>     ...print i, ' = ',sh[i]
>>> sh.close()
```

The output of this code is:

```
describe = My data
data = [1, 2, 3, 4]
map = {'x2': 200, 'x1': 100}
```

Finally, one can remove elements using the usual `del` method.

As you can see, the “shelve” module is very useful since now one can create a small persistent database to hold different Jython objects.

### 2.16.7 Using Java for I/O

In this section, we show how to write and read data by calling Java classes. Let us give an example of how to write a list of values into a binary file using the `DataOutputStream` Java class. In the example below we also use the Java class `BufferedOutputStream` to make the output operations to be more efficient. In this approach, data are accumulated in the computer memory buffer first, and are only written when the memory buffer is full.

Writing data using Java

```
from java.io import *
```

```
fo=FileOutputStream('test.d')
out=DataOutputStream(BufferedOutputStream( fo ))

list=[1.,2.,3.,4]
for a in list:
    out.writeFloat(a)

out.close()
fo.close()
```

The output of this example is binary data. The `DataOutputStream` class allows to write any of the basic types of data using appropriate methods, such as `boolean` (`writeBoolean(val)`), `double` (`writeDouble(val)`), `integers` (`writeInt(val)`), `long` (`writeLong(val)`) and so on.

Now let us read the stored float numbers sequentially. We will do this in an infinite loop using the `'while'` statement until we reach the end of the file (i.e. until the “end-line” exception is thrown). Then, the `break` statement exits the infinite loop. Since we know that our data are a sequence of float numbers, we use the method `readFloat()`. One can play with other similar methods, such as `readInt()` (read integer values), `readDouble()` (read double values).

#### Reading data using Java

```
from java.io import *

fo=FileInputStream('test.d')
inf=DataInputStream(BufferedInputStream(fo))

while 1:
    try:
        f=inf.readFloat()
        print f
    except:
        print 'end of file'
        break

inf.close()
fo.close()
```

We will continue the discussion of high-level Java classes for I/O which allow us to store objects or sequences of objects in Chap. 11.

### 2.16.8 Reading Data from the Network

Files with data may not be available from a local file storage, but exist in network-accessible locations. In this case, one should use the module `'urllib2'` that can

read data from URLs using HTTP, HTTPS, FTP file protocols. Here is an example of how to read the HTML Jython web page with Jython news:

```
>>> from urllib2 import *
>>> f = urlopen('http://www.jython.org/Project/news.html')
>>> s=f.read()
>>> f.close()
>>> print s
```

This code snippet is very similar to the I/O examples shown above, with the only one exception: now we open a file using the `urlopen` statement. The web access is unauthenticated. One can always check the response headers as `f.info()`, while the actual URL can be printed using the string `f.geturl()`. As usual, one can also use the method `readlines()` to put all HTML-page lines into a Jython list.

One can also use a `jHepWork` module for downloading files from the Web. It has one advantage: it shows a progress bar during file retrievals. This will be discussed in Sect. 12.2.

If authentication is required during file access, a client should retry the request with the appropriate name and password. The module `'urllib2'` also provides such functionality, but we will refrain from further discussion of this advanced topic.

## 2.17 Real-life Example. Collecting Data Files

Here we will consider a rather common data-analysis task: we collect all files located in a file system, assuming that all such files have the extension `'.dat'`. The files will be located in the root directory  `'/home'`, which is the usual user-home directory on the Linux/UNIX platform. Our files contain numbers, each of which is positioned on a new line. We will persuade the following task: we will try to sum up all numbers in the files and calculate the sum of all numbers inside these files.

A snippet of a module `'walker.py'` which returns a list of files is given below. The module accepts two arguments: the root directory for scanning and the extension of the files we are interested in. The function builds a list of files with the appended full path. We will call the function `walk()` recursively until all directories are identified:

```
File 'walker.py'

import os

def walker (dir,extension):
    files=[]
    def walk( dir, process):
        for f in os.listdir( dir ):
            fpath = os.path.join( dir, f)
            if os.path.isdir(fpath) and not os.path.islink(fpath):
```

```

        walk( fpath, process )
    if os.path.isfile( fpath ):
        if fpath.endswith(extension):
            files.append(fpath)

walk(dir, files)
return files

```

Let us test this module. For this, we will write a small program which: (1) imports the module 'walker.py'; (2) lists all descendant files and subdirectories under the specified directory and fills the file list with all files which have the extension '.dat'; (3) then it loops over all files in the list and reads the numbers positioned on every new line; (4) Finally, all numbers are summed up. The code which does all of this is given below:

```

_____ File collector _____

import os
from walker import *

files= walker('/home/', '.dat')

sum=0
lines=[]
for file in files:
    ifile = open(file, 'r')
    lines=lines+ifile.readlines()
    ifile.close()
    for i in range(len(lines)):
        sum=sum+float(lines[i])
print "Sum of all numbers=", sum

```

The described approach is not the only one. The module which lists all files recursively can look much sorter using the `os.walk` function:

```

_____ Building a file list _____

def getFileList(rootdir):
    fileList = []
    for root, subFolders, files in os.walk(rootdir):
        for f in files:
            fileList.append(os.path.join(root, f))
    return fileList

print getFileList('/home/')

```

This code builds a list of files in the directory "/home/".

In Sect. 12.9 we will show another efficient code based on the `jHepWork` Java class which can also be used in pure-Java applications. As in the example above, it builds a list of files recursing into all subdirectories.

The above code can significantly be simplified if we know that all input files are located inside a single directory, thus there is no need for transversing all subdirectories.

```
>>> list=[]
>>> for f in os.listdir('/home/'):
>>>     if not file.endswith('.dat'): continue
>>>     list.append(f)
```

Finally, there is a simpler approach: import the module '`glob`' and scan all files:

```
>>> import glob
>>> list=glob.glob('/home/*.dat')
```

The asterisk (\*) in this code indicates that we are searching for a pattern match, so every file or directory with the extension '`.dat`' will be put into a list, without recursing further into subdirectories. One can specify other wildcard characters, such as '`/home/data?.dat`', that matches any single character in that position in the name starting from '`data`'. Another example: '`/home/*[0-9].dat`' string considers all files that have a digit in their names before the extension '`.dat`'.

Often, in order to process data stored in many files, it is useful to divide a list with file names into several lists with equal number of files in each list. In this way, one can process files in parallel using multiple computers or multiple processors. This task can easily be achieved with the code given below:

File list splitter

```
def splitlist(seq, size):
    newlist = []
    splitsize = 1.0/size*len(seq)
    for i in range(size):
        k1=round(i*splitsize)
        k2=round((i+1)*splitsize)
        newlist.append(seq[int(k1):int(k2)])
        newlist.append(seq[k])
    return newlist
```

The code accepts a list of files and an integer `size` which specifies how many lists need to be generated. The function returns a new list in which each entry represents a list of files. The number of entries in each sublist is roughly equal.

## 2.18 Using Java for GUI Programming

Undoubtedly, the major strength of Jython is in its natural integration with Java, a language used to build Jython. This opens infinite opportunities for a programmer. Assuming that you had already a chance to look at one of these Java books [1–5], you can start immediately use Java libraries to write a Jython code.

Below we show a small example of how to write a graphical user interface which consists of a frame, a button and a text area. While the code still uses the Python syntax, it calls classes from the Java platform.

```
Swing GUI using Jython

from java.awt import *
from javax.swing import *

fr = JFrame('Hello!')
pa1 = JPanel()
pa2 = JTextArea('text', 6, 20)

def act(event):
    pa2.setText('Hello, jHepWork')

bu=JButton('Hello', actionPerformed=act)
pa1.add(bu)

fr.add(pa1, BorderLayout.SOUTH)
fr.add(pa2, BorderLayout.NORTH)
fr.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE)
fr.pack()
fr.setVisible(1)
```

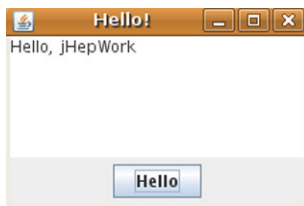
In this example, we call Java swing components directly, like they are usual Python classes. The main difference with Python is in the class names: Java classes always have names starting with capital letters.

When comparing this code with Java, one should note several important differences: there is no need to use the 'new' statement when creating Java objects. Also, there is no need to put a semicolon at the end of each Java method or class declaration. We should also recall that Java boolean values are transformed into either "1" (true) or "0" (false) in Jython programs.

So, let us continue with our example. Create a file, say, 'gui.py', copy the lines from the example below and run this file in the jHepWork editor. You will see a frame as shown in Fig. 2.1. By clicking on the button, the message "Hello, jHepWork" should be shown.

In the following chapters, we try to follow our general concept: a Jython macro is already a sufficiently high-level program style, so we will avoid detailed discussion of GUI-type of program development. In this book, we aim to show how to develop data analysis programs for which GUI-type of features are less frequent,

**Fig. 2.1** A Java Swing frame with a button “Hello”



compare to “macro”-type of programming. Since Jython macros allow manipulations with objects without dealing with low-level features of programming languages, in some sense, they are already some sort of “user-interfaces”. In addition, Jython macros have much greater flexibility than any GUI-driven application, since they can quickly be altered and rerun.

Yet, GUI is an important aspect of our life and we will discuss how to add GUI features to data-analysis applications in appropriate chapters.

## 2.19 Concluding Remarks

This concludes our introduction to the world of Python, Jython and Java. If there is one message I have tried to convey here is that the combination of all these three languages (actually, only two!) gives you an extremely powerful and flexible tool for your research. There are dozens of books written for each language and I would recommend to have some of them on your table if you want to study the topic in depth. To learn about Jython, you can always pick up a Python book (version 2.5 at least). In several cases, you may look at Jython and Java programming books, especially if you will need to do something very specific and non-standard using Java libraries. But, I almost guarantee, such situations will be infrequent if you will learn how to use the jHepWork libraries to be discussed in the following chapters.

## References

1. Richardson, C., Avondolio, D., Vitale, J., Schrager, S., Mitchell, M., Scanlon, J.: Professional Java, JDK 5th edn. Wrox, Birmingham (2005)
2. Arnold, K., Gosling, J., Holmes, D.: Java(TM) Programming Language, 4th edn. Java Series. Addison-Wesley, Reading (2005)
3. Flanagan, D.: Java in a Nutshell, 5th edn. O'Reilly Media, Sebastopol (2005)
4. Eckel, B.: Thinking in Java, 4th edn. Prentice Hall PTR, Englewood Cliffs (2006)
5. Bloch, J.: Effective Java, 2nd edn. The Java Series. Prentice Hall PTR, Englewood Cliffs (2008)



<http://www.springer.com/978-1-84996-286-5>

Scientific Data Analysis using Jython Scripting and Java

Chekanov, S.V.

2010, XXIV, 440 p., Hardcover

ISBN: 978-1-84996-286-5