Linguagens de Scripting
Passado e Futuro

FCT Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA CIÊNCIA E DO ENSINO SUPERIOR
Programa Fundo de Apoio à Comunidade Científica (FACC)

Departamento de Informática
Universidade do Minho
2004

Prefácio

As linguagens de scripting estão na moda. As razões que as levam a serem tão usadas são várias mas a maior é, sem dúvida, a flexibilidade.

Uma linguagem de scripting é uma linguagem de programação com um alto nível de abstracção, reflexiva e, regra geral, interpretada. Mas quais são as línguagens de scripting? Existem muitas, e a sua classificação não é consensual. Além das quatro que serão focadas neste volume de tutoriais (Tcl, Python, Perl e Ruby), outras que poderei classificar como tal são o Haskell, Scheme, PHP ou até a Bash.

Todas estas linguagens fazem parte de diferentes paradigmas de programação. Umas são mais orientadas a objectos, outras completamente funcionais, outras imperativas, e ainda há aquelas que são aquilo que o programador quiser. Embora diferentes, todas elas permitem uma abstração dos problemas físicos da implementação, para que se foque a implementação do problema a ser estudado.

Outro dos pontos forte destas linguagens é a sua reflexividade: permitir que em tempo de execução estas alterem o seu próprio código. Esta é, sem dúvida, uma razão que tem levado a investigação e a indústria das mais variadas áreas a adoptar estas linguagens.

Finalmente, ao serem linguagens interpretadas, permitem uma maior flexibilidade de programação, quebrando o célebre ciclo "edição", "compilação" e "execução" para apenas "edição" e "execução". Este ponto é, por vezes, apontado como negativo, devido à provável perda de eficiência. No entanto, estas linguagens conseguem ser interpretadas e eficientes mas também permitir a compilação e optimização do código.

Estas linguagens têm vindo a ser usadas dos mais variados campos da investigação, desde a Bio-Informática até ao comércio electrónico, desde a gestão de listas de correio até à geração automática de web-sites.

Para esta sessão, escolhemos quatro das linguagens mais utilizadas. O Tcl por ser uma das mais antigas, mas que não deixa de ser usada, e à qual todas as outras foram buscar inspiração. Segue-se o Perl, a mais usada linguagem de scripting e, talvez, a responsável pela expansão do uso deste tipo de linguagens nas mais variádas áreas, e o Python, uma linguagem mais estruturada do que o Perl, e muito adoptada em projectos open-source. Concluimos com o Ruby, muito menos conhecida do que estas três, mas que tem vindo a ganhar adeptos. Por ser mais recente, foi inspirar-se quer no Perl quer no Python, aproveitando desta forma os pontos fortes de cada uma delas.

Espero que esta sessão nos permita aprender e escolher a nossa linguagem predilecta. Acreditem que de todas as linguagens que possam imaginar, as de scripting são das mais úteis.

Alberto Simões

Contents

Chapter 1

Programming with Tcl

Shyam Pather

Information and Telecommunication Technology Center
University of Kansas

How to use this Document

This document is intended as an introductory tutorial to Tcl, and not as an exhaustive reference. I have attempted to highlight the most important aspects of the Tcl language, so as to provide some base material from which to get started. I have not included any material on the Tk toolkit, since there are already several Tk tutorials available on the internet. If your purpose in using Tcl/Tk is to write graphical user interfaces, then you might be tempted to rush out and learn the Tk commands and pick up the Tcl language as you go along. However, it is my belief that a solid foundation in the Tcl language is essential to writing successful Tcl/Tk code. Therefore, I would strongly recommend taking the time to go through this document and its examples *before* attempting to use Tk.

Most of the Tcl commands presented in this document can be used in many different ways, only a few of which are illustrated here. For this reason, I recommend reading the manual pages for all the commands referred to in this tutorial. For more complete and descriptive coverage of Tcl and Tk, refer to John Ousterhout's book *Tcl and the Tk Toolkit*, described on the Tcl/Tk Information page.

1.1 Introduction to Tcl

Tcl was originally intended to be a reusable command language. Its developers had been creating a number of interactive tools, each requiring its own command language. Since they were more interested in the tools themselves than the command languages they would employ, these command languages were constructed quickly, without regard to proper design.

After implementing several such "quick-and-dirty" command languages and experiencing problems with each one, they decided to concentrate on implementing a general-purpose, robust command language that could easily be integrated into new applications. Thus Tcl (Tool Command Language) was born.

Since that time, Tcl has been widely used as a scripting language. In most cases, Tcl is used in combination with the Tk ("Tool Kit") library, a set of commands and procedures that make it relatively easy to program graphical user interfaces in Tcl.

One of Tcl's most useful features is its extensibility. If an application requires some functionality not offered by standard Tcl, new Tcl commands can be implemented using the C language, and integrated fairly easily. Since Tcl is so easy to extend, many people have written extension packages

for some common tasks, and made these freely available on the internet. (For more information, see the Tcl/Tk Information page).

1.2 Tcl Programming Basics

The main difference between Tcl and languages such as C, is that Tcl is an *interpreted* rather than a *compiled* language. Tcl programs are simply scripts consisting of Tcl commands that are processed by a Tcl interpreter at run time. One advantage that this offers is that Tcl programs can themselves generate Tcl scripts that can be evaluated at a later time. This can be useful, for example, when creating a graphical user interface with a command button that needs to perform different actions at different times.

The next several sections descibe the essential elements of Tcl programs. Each section is accompanied by a series of examples, and a sample Tcl interpreter that you can be use to try out the examples yourself.

1.2.1 Variables and Variable Substitution

Variables in Tcl, as in most other languages, can be thought of as boxes in which various kinds of data can be stored. These boxes, or variables, are given names, which are then used to access the values stored in them.

Unlike C, Tcl does not require that variables be declared before they are used. Tcl variables are simply created when they are first assigned values, using the set command. Although they do not have to be deleted, Tcl variables can be deleted using the unset command.

The value stored in a variable can be accessed by prefacing the name of the variable with a dollar sign ("$"). This is known as variable substitution, and is illustrated in the examples below.

Tcl is an example of a "weakly typed" language. This simply means that almost any type of data can be stored in any variable. For example, the same variable can be used to store a number, a date, a string, or even another Tcl script.

Example 1.1

```
Code:
    set foo "john"
    puts "Hi my name is $foo"

Output:
    Hi my name is john
```

Example 1.1 illustrates the use of variable substitution. The value "john" is assigned to the variable "foo", whose value is then substituted for "$foo". Note that variable substitution can occur within a string. The puts command (described in a later section) is used to display the string.

Example 1.2

```
Code:
    set month 2
    set day 3
    set year 97
    set date "$month:$day:$year"
    puts $date

Output:
    2:3:97
```

Here variable substitution is used in several places: The values of the variables "month", "day", and "year" are substituted in the set command that assigns the value of the "date" variable, and the value of the "date" variable is then substituted in the line that displays the output.

Example 1.3

```
Code:
    set foo "puts hi"
    eval $foo

Output:
    hi
```

In this example, the variable "foo" holds another (small) Tcl script that simply prints the word "hi". The value of the variable "foo" is subsituted into an eval command, which causes it to be evaluated by the Tcl interpreter (the eval command will be described in greater detail in a later section).

1.2.2 Expressions

Tcl allows several types of expressions, including mathematical expressions, and relational expressions. Tcl expressions are usually evaluated using the expr command, as illustrated in the examples below.

Example 2.1

```
Code:
    expr 0 == 1

Output:
    0
```

Example 2.2

```
Code:
    expr 1 == 1

Output:
    1
```

Examples 2.1 and 2.2 illustrate the use of relational expressions with the expr command. The first expression evaluates to 0 (false) since 0 does not equal 1, whereas the second expression evalutates to 1 (true), since, obviously, 1 does equal 1. The relational operator "==" is used to do the comparison.

Example 2.3

```
Code:
    expr 4 + 5

Output:
    9
```

Example 2.3 shows how to use the expr statement to evaluate an arithmetic expression. Here the result is simply the sum of 4 and 5. Tcl offers a rich set of arithmetic and relational operators, each of which is described in the expr manual page.

Example 2.4

```
Code:
    expr sin(2)

Output:
    0.909297
```

This example shows that the expr statement can be used to evaluate the result of a mathematical function, in this case, the sine of an angle. Tcl offers many such mathematical functions, also described on the expr manual page.

1.2.3 Command Substitution

Just as variable substitution is used to substitute the value of a variable into a Tcl script, command substitution can be used to replace a Tcl command with the result it returns. Consider the following example:

Example 3.1

```
Code:
    puts "I am [expr 10 * 2] years old, and my I.Q. is [expr 100 - 25]"

Output:
    I am 20 years old, and my I.Q. is 75
```

As this example shows, square brackets are used to achieve command substitution: The text between the square brackets is evaluated as a Tcl script, and its result is then substituted in its place. In this case, command substitution is used to place the results of two mathematical expressions into a string. Command substitution is often used in conjunction with variable substitution, as shown in Example 3.2:

Example 3.2

```
Code:
    set my_height 6.0

    puts "If I was 2 inches taller, I would be
          [expr $my_height + (2.0 / 12.0)] feet tall"

Output:
    If I was 2 inches taller, I would be 6.16667 feet tall
```

In this example, the value of the variable "my_height" is substituted inside the angle brackets before the command is evaluated. This is a good illustration of Tcl's one-pass recursive parsing mechanism. When evaluating a statement, the Tcl interpreter, makes one pass over it, and in doing so makes all the necessary substitutions. Once this is done, the interpreter then evaluates the resulting expression. If, during its pass over the expression, the interpreter encounters square brackets (indicating that command substitution is to be performed), it recursively parses the script inside the square brackets in the same manner. For more information on one-pass parsing, refer to Matt Peters' document on the topic.

1.2.4 Control Flow

In all but the simplest scripts, some mechanism is needed to control the flow of execution. Tcl offers decision-making constructs (if-else and switch statements) as well as looping constructs (while, for, and foreach statements), both of which can alter the flow of execution in response to some condition. The following examples serve to illustrate these constructs.

Example 4.1

```
Code:
    set my_planet "earth"

    if {$my_planet == "earth"} {
        puts "I feel right at home."
    } elseif {$my_planet == "venus"} {
        puts "This is not my home."
```

```
    } else {
        puts "I am neither from Earth, nor from Venus."
    }

    set temp 95
    if {$temp < 80} {
        puts "It's a little chilly."
    } else {
        puts "Warm enough for me."
    }

Output:
    I feel right at home.
    Warm enough for me.
```

Example 4.1 makes two uses of the if-statement. It sets the value of the variable "my_planet" to "earth", and then uses an if-statement to choose which statement to print. The general syntax of the if-statementis as follows:

`if test1 body1 ?elseif test2 body2 elseif ...? ?else bodyn?`

If the test1 expression evaluates to a true value, then body1 is executed. If not, then if there are any elseif clauses present, their test expressions are evalutated and, if true, their bodies are executed. If any one of the tests is made successfully, after its corresponding body is executed, the if-statement terminates, and does not make any further comparisons. If there is an else clause present, its body is executed if no other test succeeds.

Another decision-making construct is the switch-statement. It is a simplification of the if-statement that is useful when one needs to take one of several actions depending on the value of a variable whose possible values are known. This is illustrated in Example 4.2, which uses a switch statement to print a sentence, depending on the value of a variable "num_legs".

Example 4.2

```
Code:
    set num_legs 4
    switch $num_legs {
         2 {puts "It could be a human."}
         4 {puts "It could be a cow."}
         6 {puts "It could be an ant."}
         8 {puts "It could be a spider."}
         default {puts "It could be anything."}
    }

Output:
    It could be a cow.
```

The switch-statement has two general forms (both of which are described in detail in the manual page), but the form used here is as follows:

`switch ?options? string {pattern body ?pattern body ...?}`

Basically, the string argument is compared to each of the patterns and if a comparison succeeds, the corresponding body is executed, after which the switch statement returns. The pattern "default", if present, is always matched, and thus its body always executed if none of the earlier comparisons succeed.

It is often useful to execute parts of a program repeatedly, until some condition is met. In order to facilitate this, Tcl offers three looping constructs: the while, for, and foreach statements, each of which is shown in the examples below.

Example 4.3

```
Code:
    for {set i 0} {$i < 10} {incr i 1} {
        puts "In the for loop, and i == $i"
    }

Output:
    In the for loop, and i == 0
    In the for loop, and i == 1
    In the for loop, and i == 2
    In the for loop, and i == 3
    In the for loop, and i == 4
    In the for loop, and i == 5
    In the for loop, and i == 6
    In the for loop, and i == 7
    In the for loop, and i == 8
    In the for loop, and i == 9
```

The general syntax for the for-loop is as follows:

`for init test reinit body`

The init argument is a Tcl script that initializes a looping variable. In the for-loop used in Example 4.3, the looping variable was called "i", and the init argument simply set it to 0. The test argument is a Tcl script which will be evaluated to decide whether or not to enter the body of the for-loop. Each time this script evaluates to a true value, the body of the loop is executed. The first time this script evaluates to false, the loop terminates. The reinit argument specifies a script that will be called after each time the body is executed. In Example 4.3, the reinit script increments the value of the looping variable, "i". Thus, for-loop in this example executes its body 10 times, before its test script evaluates to false, causing the loop to terminate.

Example 4.4

```
Code:
    set i 0
    while {$i < 10} {
        puts "In the while loop, and i == $i"
        incr i 1
    }

Output:
    In the while loop, and i == 0
    In the while loop, and i == 1
    In the while loop, and i == 2
    In the while loop, and i == 3
    In the while loop, and i == 4
    In the while loop, and i == 5
    In the while loop, and i == 6
    In the while loop, and i == 7
    In the while loop, and i == 8
    In the while loop, and i == 9
```

Example 4.4 illustrates the use of a while-loop, the general syntax of which follows the form:

`while test body`

The basic concept behind the while-loop is that while the script specified by the test argument evaluates to a true value, the script specified by the body argument is executed. The while loop in Example 4.4 accomplishes the same effect as the for-loop in Example 4.3. A looping variable,

"i", is again initialized to 0 and incremented each time the loop is executed. The loop terminates when the value of "i" reaches 10.

Note, that in the case of the while-loop, the initialization and re-initialization of the looping variable are not part of the while-statement itself. Therefore, the initialization of the variable is done before the while-loop, and the reinitialization is incorporated into its body. If these statements were left out, the code would probably still run, but with unexpected results.

Example 4.5

```
Code:
    foreach vowel {a e i o u} {
        puts "$vowel is a vowel"
    }

Output:
    a is a vowel
    e is a vowel
    i is a vowel
    o is a vowel
    u is a vowel
```

The foreach-loop, illustrated in Example 4.5, operates in a slightly different manner to the other types of Tcl loops described in this section. Whereas for-loops and while-loops execute while a particular condition is true, the foreach-loop executes once for each element of a fixed list. The general syntax for the foreach-loop is:

```
foreach varName list body
```

The variable specified by varName takes on each of the values in the list in turn, and the body script is executed each time. In Example 4.5, the variable "vowel" takes on each of the values in the list "{a e i o u}" (Tcl list structure will be discussed in more detail in a later section), and for each value, the body of the loop is executed, resulting in one printed statement each time.

1.2.5 Procedures

Procedures in Tcl serve much the same purpose as functions in C. They may take arguments, and may return values. The basic syntax for defining a procedure is:

```
proc name argList body
```

Once a procedure is created, it is considered to be a command, just like any other built-in Tcl command. As such, it may be called using its name, followed by a value for each of its arguments. The return value from a procedure is equivalent to the result of a built-in Tcl command. Thus, command substitution can be used to substitute the return value of a procedure into another expression.

By default, the return value from a procedure is the result of the last command in its body. However, to return another value, the return command may be used. If an argument is given to the return command, then the value of this argument becomes the result of the procedure. The return command may be used anywhere in the body of the procedure, causing the procedure to exit immediately.

Example 5.1

```
Code:
    proc sum_proc {a b} {
        return [expr $a + $b]
    }

    proc magnitude {num} {
        if {$num > 0} {
            return $num
        }
```

```
        set num [expr $num * (-1)]
        return $num
    }

    set num1 12
    set num2 14
    set sum [sum_proc $num1 $num2]

    puts "The sum is $sum"
    puts "The magnitude of 3 is [magnitude 3]"
    puts "The magnitude of -2 is [magnitude -2]"

Output:
    The sum is 26
    The magnitude of 3 is 3
    The magnitude of -2 is 2
```

This example first creates two procedures, "sum_proc" and "magnitude". "sum_proc" takes two arguments, and simply returns the value of their sum. "magnitude" returns the absolute value of a number. After the procedure definitions, three global variables are created. The last of these, "sum" is assigned the return value of the procedure "sum_proc", called with the values of the variables "num1" and "num2" as arguments. The "magnitude" procedure is then called twice, first with "3" as an argument, then with "-2".

The "sum_proc" procedure uses the expr command to calculate the sum of its arguments. The result of the expr command is substituted into the return statement, making it the return value for the procedure.

The "magnitude" procedure makes use of an if-statement to take different actions, depending on the sign of its argument. If the number is postive, its value is returned, and the procedure exits immediately, skipping all the rest of its code. Otherwise, the number is multiplied by -1 to obtain its magnitude, and this value is returned. The same effect could be achieved by moving the statement that multiplies the value by -1 into an else-clause, but the purpose of this example was to illustrate the use of the return statement at several locations within a procedure.

Inside the body of a procedure, new variables may be created with the set command as normal. However, these variables will be local to the procedure, and will no longer be accessible once the procedure returns. If access to global variables is needed inside a procedure, these may be accessed by means of the global keyword, as described in Example 5.2.

Example 5.2

```
Code:
    proc dumb_proc {} {
        set myvar 4
        puts "The value of the local variable is $myvar"

        global myglobalvar
        puts "The value of the global variable is $myglobalvar"
    }

    set myglobalvar 79
    dumb_proc

Output:
    The value of the local variable is 4
    The value of the global variable is 79
```

The procedure "dumb_proc" achieves no special purpose, and is simply designed to illustrate the use of the global keyword to access global variables. It takes no arguments, and as such its argument list is empty. Note that even though the procedure takes no arguments, the empty list structure must still be included.

The procedure first creates a local variable, "myvar", sets its value to "4", and then displays it. Then it uses the global keyword to gain access to a global variable named "myglobalvar". The value of this global variable is then printed.

After the procedure definition, a global variable "myglobalvar" is created, and assigned a value of "79". The procedure "dumb_proc" is then called, resulting in the output shown above.

1.2.6 Lists

Lists in Tcl provide a simple means by which to group collections of items, and deal with the collection as a single entity. When needed, the single items in the group can be accessed individually. Lists are represented in Tcl as strings with a specified format. As such, they can be used in any place where strings are normally allowed. The elements of a list are also strings, and therefore any form of data that can be represented by a string can be included in a list (allowing lists to be nested within one another).

The following examples will illustrates many important list commands:

Example 6.1

```
Code:
    set simple_list "John Joe Mary Susan"
    puts [lindex $simple_list 0]
    puts [lindex $simple_list 2]

Output:
    John
    Mary
```

Example 6.1 creates a simple list of four elements, each of which consists of one word. The lindex command is then used to extract two of the elements in the list: the 0th element and the 2nd element. Note that list indexing is zero-based. It is also important to see that the lindex command, along with most other list commands, takes an actual list as its first argument, not the name of a variable containing a list. Thus the value of the variable "simple_list" is substitued into the lindex command.

Example 6.2

```
Code:
    set simple_list2 "Mike Sam Heather Jennifer"
    set compound_list [list $simple_list $simple_list2]
    puts $compound_list
    puts [llength $compound_list]

Output:
    {John Joe Mary Susan} {Mike Sam Heather Jennifer}
    2
```

Example 6.2 is a continuation of Example 6.1, and assumes the variable "simple_list" (created in Example 6.1) still exists. In this example, a new variable called "simple_list2" is created, and assigned the value of another simple four-element list. A compound list is then formed by using the list command, which simply forms a list from its arguments. The list command ensures that proper list structure is observed, even when its arguments themselves are lists, or other complex structures. Displaying the value of "compound_list" shows that it is a list of two elements, each of which is itself a list of four elements. The llength command is used to obtain the length of the list, "compund_list", which is 2 in this case.

This example highlights two ways in which to create lists in Tcl: by explicitly listing the elements within quotes, and by using the list command. Explicity listing the elements works well when each of the elements is a single word. However, if the elements contain whitespaces, then maintaining proper list structure becomes a little more tricky. For these cases, the list command proves very useful.

Example 6.3

```
Code:
    set mylist "Mercury Venus Mars"
    puts $mylist
    set mylist [linsert $mylist 2 Earth]
    puts $mylist
    lappend mylist Jupiter
    puts $mylist

Output:
    Mercury Venus Mars
    Mercury Venus Earth Mars
    Mercury Venus Earth Mars Jupiter
```

In example 6.3, a simple list of 3 items is created, and assigned to the variable "mylist". The linsert command is then used to insert a new item into this list. Note that, as with the llength command, the linsert command takes an actual list as its first argument, not the name of a variable containing a list. The linsert command returns a list that is the same as the list it was passed, except that the specified item is inserted in the appropriate position. This return value needs to be assigned back to the variable "mylist" in order for the list stored in that variable to change.

One list command that does not behave in this way is the lappend command. It takes the name of a variable as its first argument, and appends its subsequent arguments onto the list stored in that variable. Thus the value of the variable is modified directly. Understanding the difference between the way the lappend command works, and the way that commands such as linsert work is fundamental to using lists correctly.

The list commands presented here are only a small subset of those available. Refer to the manual pages, or one of the other Tcl/Tk references for a complete description of all list commands.

1.2.7 Arrays

Another way of grouping data in Tcl is to use arrays. Arrays are simply collections of items in which each item is given a unique index by which it may be accessed. As with all other Tcl variables, arrays need not be declared before they are used, and, unlike arrays in C, their size need not be specified either.

An individual element of an array may be referred to by using the array name, followed immediately by the index of the element, enclosed in parentheses. Array elements are treated much like any other Tcl variables. They are created by means of the set command, and their values can be substituted using the dollar sign ("$"), as is the case with other variables.

Example 7.1

```
Code:
    set myarray(0) "Zero"
    set myarray(1) "One"
    set myarray(2) "Two"

    for {set i 0} {$i < 3} {incr i 1} {
        puts $myarray($i)
    }

Output:
```

```
    Zero
    One
    Two
```

In Example 7.1, an array called "myarray" is created and initialized. Note that no special code is required to create the array because it is created by the set statement that assigns a value to its first element. The for-loop simply prints out the value stored in each element of the array. Note the use of variable substitution in the array index and the array name.

Example 7.2

```
Code:
    set person_info(name) "Fred Smith"
    set person_info(age) "25"
    set person_info(occupation) "Plumber"

    foreach thing {name age occupation} {
        puts "$thing == $person_info($thing)"
    }

Output:
    name == Fred Smith
    age == 25
    occupation == Plumber
```

Example 7.2 illustrates one of the unique features of Tcl arrays: array indices need not be integers. In fact, array indices can take on any string value. In this case, the array "person_info" is created with three elements. The indices for the elements are "name", "age", and "occupation". The foreach-loop simply displays each of the elements in the array.

Using arrays with named indices is one of the ways to abstract objects in Tcl. In Example 7.2, the "person_info" array can be thought of as an "object" describing a person. Each of the elements in the array then describes a fundamental attribute of the object.

One problem with using named indices with arrays is that one needs to remember the names of all the elements in order to traverse the array. In Example 7.2, for example, the names of all the elements had to be listed explicitly. In a case such as this one, in which there are only three elements, this does not present much of a problem. However, if the array contained many more elements, explicitly listing them each time the array had to be traversed would lead to very messy code. The array Tcl command, illustrated in Example 7.3, provides a means to get around this problem.

Example 7.3

```
Code:
    set person_info(name) "Fred Smith"
    set person_info(age) "25"
    set person_info(occupation) "Plumber"

    foreach thing [array names person_info] {
        puts "$thing == $person_info($thing)"
    }

Output:
    occupation == Plumber
    age == 25
    name == Fred Smith
```

Example 7.3 produces essentially the same result as Example 7.2, but it makes use of the array command to obtain the names of the elements in the array, instead of listing them explicitly. The

array elements are displayed in a different order than they were in Example 7.2, simply because the array command returns the names of the elements in a different order than the one in which they were explicitly listed previously.

The general purpose of the array command is to retrieve various pieces of information about an array (such as its size or the names of its elements), and perform other operations (such as searching) on it. The general syntax of the array command is:

```
array option arrayName ?arg arg ...?
```

The option argument specifies which array operation to perform. In the case of Example 7.3, the option argument is given the value "names", which causes the array command to return a list of the names of the elements in the array given by the arrayName argument. For a complete list of the allowed values of the option argument, and well as a description of the corresponding operations, refer to the manual page for the array command.

1.2.8 Strings

Since strings are the most prevalent data type in Tcl, it makes sense that Tcl provides a rich set of functions for manipulating them. Most string operations are done by means of the string command, which takes the following general form:

```
string option arg ?arg ...?
```

The string command actually performs several different functions, and the option argument is used to differentiate between them. Example 8.1 creates a string and then uses the string command to manipulate it, and obtain information about it.

Example 8.1

```
Code:
    set str "This is a string"

    puts "The string is: $str"
    puts "The length of the string is: [string length $str]"
    puts "The character at index 3 is: [string index $str 3]"
    puts "The characters from index 4 through 8 are: [string range $str 4 8]"
    puts "The index of the first occurrence of letter \"i\" is:
          [string first i $str]"

Output:
    The string is: This is a string
    The length of the string is: 16
    The character at index 3 is: s
    The characters from index 4 through 8 are: is a
    The index of the first occurrence of letter ‘‘i’’ is: 2
```

In Example 8.1, a variable called "str" is created, and initialized to the value, "This is a string". The string commmand is then used with various options to obtain various pieces of information about the string. Refer to the manual page for the string command for a complete listing and explanation of the various options. Also, there are several other string-related commands worth exploring, such as format, regexp, regsub, and scan.

1.2.9 Input/Output

Most input and output operations in Tcl are done by means of the puts and gets commands. Most of the examples in this document have made use of the puts command to display output on the console. In a similar manner, the gets command can be used to wait for input from the console, and optionally store it an a variable. Its general syntax has the following form:

```
gets channelId ?varName?
```

The first argument to gets is the name of an open channel from which to read data, and can be thought of as a *file descriptor* in the C sense. If the varName argument is specified, gets stores the data it reads in that variable, and returns the number of bytes read. If varName is not specified, then gets simply returns the data it read.

Example 9.1

```
Code:
    puts -nonewline "Enter your name: "
    set bytesread [gets stdin name]

    puts "Your name is $name, and it is $bytesread bytes long"

Output: (note that user input is shown in italics)
    Enter your name: Shyam
    Your name is Shyam, and it is 5 bytes long
```

Example 9.1 makes use of both the puts and gets commands. The puts command is used with the -nonewline flag to suppress the trailing newline that it normally appends to its output. A variable, "bytesread", is then assigned the result of a gets command that reads from the channel "stdin" (the standard input), and stores the data it reads in the variable, "name". Thus "bytesread" ends up storing the number of bytes of user input read from the console.

In Example 9.1, gets was used to read from the channel "stdin" (created automatically when the Tcl interpreter is started) which corresponds to the standard input. The puts command can also be used with a channel identifier to write to a specific channel. However, if no channel identifier is passed to puts, it writes to the standard output (this is the way puts has been used throughout this document). In addition to the standard input and output, channels can also be created to read from other types of files. As illustrated by Example 9.2, the open command can be used to open a channel to a file, and obtain an appropriate identifier for the channel. This identifier can then be passed to gets to read from the file, or puts to write to the file.

Example 9.2

```
Code:
    set f [open "/tmp/myfile" "w"]

    puts $f "We live in Texas. It's already 110 degrees out here."
    puts $f "456"

    close $f

Output: (none)
```

This example uses the open command to open a channel to a file called "/tmp/myfile". The syntax of the open command can take on three forms, one of which is:

```
open name ?access?
```

The access argument specifies what type of access (for example, read-only access or read-write access) to the file given by name is desired. See the manual page for the open command for a complete description of the access modes. In this case, write-only access is desired, so the value "w" is given for the access argument.

The open command returns a channel identifier that can be used with gets and puts to read and write from the file. In Example 9.2, this identifier is stored in the variable, "f". The puts command is then used to write two strings to the file, and then the close command is used to close the file. Example 9.3 reads the file created in Example 9.2, and displays its contents.

Example 9.3

```
Code:
    set f [open "/tmp/myfile" "r"]
```

```
    set line1 [gets $f]
    set len_line2 [gets $f line2]

    close $f

    puts "line 1: $line1"
    puts "line 2: $line2"
    puts "Length of line 2: $len_line2"

Output:
    line 1: We live in Texas. It's already 110 degrees out here.
    line 2: 456
    Length of line 2: 3
```

The file, "/tmp/myfile", is opened in read-only mode with the open command. The gets command is then used with the channel identifier returned by open to read from the file. The first call to gets does not give it the name of a variable in which to store the data it reads, so this data is returned instead. Command substitution is used to store it in the variable, "line1". The second call to gets tells it to store the data it reads in the variable, "line2". Therefore, gets would return the number of bytes it read, which, by means of command substitution, is stored in the variable "len_line2". Since all the data has been read, the file is then closed.

In this case, it was known that the file contained only two lines of data. If the length of the file was not known, the eof command could be used with a while loop to read until the end of the file was reached.

1.2.10 Other Miscellaneous Tcl Commands

eval

As described earlier, Tcl uses a one-pass parsing mechanism when evaluating scripts. It is sometimes useful, however, to have the interpreter make more than one pass over a script before evaluating it. Being able to force the interpreter to parse a script more than once allows one to store Tcl scripts in variables, and have them be evaluated at a later time. This is shown in Example 10.1:

Example 10.1

```
Code:
    set foo "set a 22"
    eval $foo
    puts $a

Output:
    22
```

The variable "foo" is set to the value "set a 22", which is itself a Tcl script. Next, the value of the variable "foo" is substituted into the eval command. The eval command simply passes its arguments through the Tcl interpreter for another round of parsing. When the interpreter encounters the statement "eval $foo", the first round of parsing simply substitutes the value of the variable "foo" in the place of "$foo", resulting in the expression "eval set a 22". The eval command then sends its arguments, "set a 22", through the interpreter again, resulting in the variable "a" being created and assigned the value "22".

One might be tempted to think that the use of the eval command could be avoided and simply replaced with the statement, `$foo`

This does not work because, on encountering the statement "$foo", the interpreter simply replaces it with the value stored in the variable "foo", and then considers its parsing work done. So, "$foo" gets replaced by "set a 22", but the interpreter never parses "set a 22", which it needs to do to make sense of the components of the statement (it needs to realize that "set" corresponds to a built in command, and that it is being passed two arguments, "a" and "22") and evaluate it correctly .

catch

When an error occurs in a Tcl command, the entire script of which it is a part is halted, and an error message is displayed. However, instead of halting the whole Tcl script, it may be useful to simply display a friendly error message and continue execution of the Tcl script.

The catch command prevents Tcl's error handling mechanisms from executing (and thus halting execution) and simply returns a meaningful value when an error occurs. This allows the program to define its own behaviour in the case of an error.

Example 10.2

```
Code:
    set retval [catch {set f [open "nosuchfile" "r"]}]

    if {$retval == 1} {
        puts "An error occured"
    }

Output: (this output occurs if there is no file named
         ‘‘nosuchfile’’ in the current directory).
    An error occured
```

The catch command is given a Tcl script as an argument. It evaluates this script, and if an error occurs, it returns 1, otherwise it returns 0. In Example 10.2, the script passed to catch tries to open a file named "nosuchfile". Assuming that no file with this name exists in the current directory, the open command should return an error. Since it occurs within a catch statement, the normal Tcl error handling routines do not get invoked, and the catch command simply returns 1. This return value is assigned to the variable "retval", which is checked to determine whether or not to print the error message. The catch command can be used in many different ways, only one of which is shown here. Refer to the manual page for a more complete description.

Chapter 2

The Long and the Short of Perl

Marty Pauley
marty@kasei.com

Preface

Scripting Language often sounds like an insult, especially when said by some C++, C#, and Java programmers. They imply that scripting languages are limited in some way.

For languages like Perl, Python, Ruby, and TCL, *high-level language* is a better description. These languages are much more expressive than low-level languages like C++, C# and Java.

2.1 What is Perl?

```
perldoc perlintro
```

> Perl is a general-purpose programming language originally developed for text manipulation and now used for a wide range of tasks including system administration, web development, network programming, GUI development, and more.
>
> ...
>
> The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal). Its major features are that it's easy to use, supports both procedural and object-oriented (OO) programming, has powerful built-in support for text processing, and has one of the world's most impressive collections of third-party modules.
>
> ...
>
> Different definitions of Perl are given in *perl*, *perlfaq1* and no doubt other places. From this we can determine that Perl is different things to different people, but that lots of people think it's at least worth writing about.

```
perldoc perlfaq1
```

> Perl is a high-level programming language with an eclectic heritage written by Larry Wall and a cast of thousands. It derives from the ubiquitous C programming language and to a lesser extent from sed, awk, the Unix shell, and at least a dozen other tools and languages. Perl's process, file, and text manipulation facilities make it particularly well-suited for tasks involving quick prototyping, system utilities, software tools,

> system management tasks, database access, graphical programming, networking, and world wide web programming. These strengths make it especially popular with system administrators and CGI script authors, but mathematicians, geneticists, journalists, and even managers also use Perl. **Maybe you should, too.**

Perl is not the most popular, most beautiful, or most efficient programming language in the world, but it is **probably the most useful**.

2.2 Long Perl

Some people think Perl can't be used for large systems. Sadly, some of these people actually work as Perl programmers. But they're wrong! Perl is an excellent language for large and complex systems.

2.2.1 Object orientation

Object orientation is still the current paradigm of choice for writing large programs; Perl supports OO. Normal Perl OO is class-based and allows multiple inheritance. Perl actually allows you to use many different forms of object orientation, if you want.

Classes in Perl are not closed. It's easy to add a new method to an existing class, without even modifying the module file. You can even add methods to the UNIVERSAL class.

```
sub UNIVERSAL::moniker {
    (ref( $_[0] ) || $_[0]) =~ /([^:]+)$/;
    return lc $1;
}
```

2.2.2 Other paradigms

Perl supports object orientation but **doesn't force you to use it**. Perl also supports procedural and functional paradigms. You can choose whatever style suits your problem.

2.2.3 Higher-level programming

Perl allows you to write code that writes code. The most obvious way is to use the `eval` function, but there are even more elegant ways.

2.2.4 Lexical closures

A lexical closure is a subroutine that refers to variables in the lexical scope where it was created. It's surprising how useful such a simple construct can be.

```
sub iterator {
    my @data = @_;
    return sub { shift @data }
}
```

2.2.5 Symbol table access

Perl package namespaces can be accessed from within the program: that's how the `import` subroutine works. You can examine and modify namespaces easily.

```
no warnings 'redefine';
local *MIME::Lite::send = sub {
    warn "Sending email\n" };
```

That can be useful while testing.

```
sub trace {
    my $victim = shift;
    no strict 'refs';
    no warnings 'redefine';
    my $orig = *{$victim}{CODE};
    *{$victim} = sub {
        warn "calling $victim(@_)\n";
        goto $orig; };
}
trace('MIME::Lite::send');
```

2.3 Short Perl

Perl is an excellent language for *short* programs. Perl doesn't force you to write "scaffolding" code. Perl will even write some code for you, if you ask.

2.3.1 Running Perl

We know that Perl code can live in files, but it doesn't have to. You can run short Perl programs directly from the command line by using the `-e` switch.

```
perl -e 'print "Hello World\n"'
```

Useful Perl programs can be so short that you can easily write them each time you need them.

2.3.2 cat

Implementing the Unix `cat` program in Perl is not difficult.

```
perl -e 'while (<>) {print}' *.txt
```

But you can shorten that.

```
perl -ne 'print' *.txt
```

The `-n` switch wraps your code in an input loop, so the running code is the same in both examples above.

Cheshire cat

If that wasn't short enough, you can remove all your code.

```
perl -pe '' *.txt
```

The `-p` option wraps your code in an input loop like `-n`, but also adds a `print` in every iteration.

```
while (<>) {} continue {print}
```

2.3.3 **** off, cat!

There is no good reason to reimplement `cat` in Perl, unless you're using a Windows machine. But you might want a *smart* `cat` that will replace all the offensive words with asterisks.

```
perl -MRegexp::Common -pe 's/$RE{profanity}/****/g' *.txt
```

Keeping it clean

You might want to permanently remove those offensive words from your files. Perl makes that easy too.

```
perl -MRegexp::Common -i -pe 's/$RE{profanity}/****/g' *.txt
```

The `-i` option performs *in-place* editing.

2.4 CPAN

The Comprehensize Perl Archive Network is a repository of modules for every task you can imagine, and then some more! Have a look at `http://search.cpan.org/`

You don't need to write the code if someone else already has. CPAN is why *you* should use Perl.

2.4.1 A simple class

In many other OO languages even simple classes require more code than they should.

A simple class example that is frequently used is an "employee" class. An employee has attributes for his name, tax reference, and annual salary.

With `Class::Accessor` from CPAN we could write:

```
package Employee;
use base 'Class::Accessor';
__PACKAGE__->mk_accessors(
    qw(name taxref salary));
```

Then you can do

```
my $worker = Employee->new({ name => 'Fred', taxref => 'N1234B' });
$worker->salary(20000);

print $worker->name, " earns ", $worker->salary, " euro\n";
```

Alternativly, if your Employee class isn't going to get any more complicated, you could use `Class::Struct` instead:

```
use Class::Struct;
struct Employee => { name => '$', taxref => '$', salary => '$' };
```

2.4.2 Database access

We have a MySQL movie database containing a "film" table with information about specific films, like the title, genre, year of release, and plot outline.

We need to implement a `Film` class that will allow us to

- search for specific films
- update the information for a film
- add new films
- remove existing films

Our *complete* film class would look like this:

```
package Film;
use base 'Class::DBI::mysql';
__PACKAGE__->set_db('Main',
    'dbi:mysql:movies',
    'user', 'password');
__PACKAGE__->set_up_table("film");
```

Then you can do

```
my $x2 = Film->create({
    title => 'X-Men 2',
    genre => 'sci-fi'});

$x2->plot('good mutants fight baddies');
$x2->update;

my @sf = Film->search(genre => 'sci-fi');
```

2.4.3 CGI script

Now that we have a Film class, it would be great to have a simple CGI script to display a list of films. When a film is selected from the list, the script should display all the details for that film.

```
use CGI ':standard';
print header;
if (my $id = param('id')) {
    show_film($id);
} else {
    show_list();
}

sub show_film {
    my $film = Film->retrieve($_[0]);
    print start_html($film->title),
        h1($film->title), p($film->plot),
        p($film->genre, $film->release),
        end_html;
}

sub show_list {
    print start_html("films"),
        h1("all films"),
        ul( map li(
            a({href=>"?id=$_"}, $_->title)),
            Film->retrieve_all),
```

```
        end_html;
}
```

2.4.4 WWW client

It would be great if we could use our offensive word elimator on the WWW.

```
use LWP::Simple;
use Regexp::Common;
my $page = get($url);
$page =~ s/$RE{profanity}/****/g;
print $page;
```

Another WWW client

Perl doesn't enforce any particular moral standards...

```
use Acme::PrOn::Automate qw(:categories);
my $naughty = Acme::PrOn::Automate->new(
    sources => [qw(Free6 Easypic)],
    categories => [BABES, LINGERIE],
    db => "naughty_db",
);
$naughty->fetch();
```

2.4.5 Email

Back to our quest to remove offensive words. This time, let's clean up our email. We can start by filtering offensive email.

```
use Mail::Audit;
use Regexp::Common;
my $msg = Mail::Audit->new();
if (grep /$RE{profanity/, @{$msg->body}) {
    $mail->reject("offensive email");
}
$msg->accept("inbox");
```

Now check email received before we started filtering.

```
use Mail::Box::Manager;
use Regexp::Common;
my $mgr = Mail::Box::Manager->new;
my $box = $mgr->open(folder => $ENV{MAIL});
for my $msg ($box->messages) {
    $msg->delete
        if $msg->string =~ /$RE{profanity/;
}
$box->close;
```

2.4.6 Box, Ox, Octopus, and Sheep

English has some obscure rules to make plurals, and Perl knows them.

```
use Lingua::EN::Inflect 'PL';
for (qw/box ox octopus sheep/) {
    print "$_ => ", PL($_), "\n";
}
```

2.4.7 Inline

"I have to use a particular Java class." Again, Perl doesn't enforce any moral standards.

```
package Searcher;
use Inline Java => 'STUDY',
    STUDY => [qw( com.kasei.Lucy )],
    SHARED_JVM => 1;
my $jc = Searcher::com::kasei::Lucy->new;
$jc->search("scripting language");
```

2.4.8 Other useful modules

- `AI::NeuralNet::Simple`
- `Algorithm::Diff`
- `Bioperl`
- `File::Slurp`
- `POE`
- `Spreadsheet::WriteExcel::Simple`
- `Template` Toolkit

2.5 More information

Perl comes with lots of documentation. `perldoc perl` will give you a list of other documents to read, both tutorial and reference.

- `http://perl.com/` sometimes has good articles.
- `http://perl.org/` has links to a wide range of Perl stuff.

2.5.1 YAPC::Europe

Yet Another Perl Conference for Europe will be in Belfast in September 2004. Many of the best Perl Hackers will be there. Admission is 99 Euro for 3 days. `http://belfast.yapc.org/` for more details.

Chapter 3

Python na Prática

Christian Robottom Reis
Async Open Source
kiko@async.com.br

3.1 Introdução

Este tutorial foi criado para apoiar um curso básico da linguagem de programação Python. Em especial, ele tem duas características: primeiro, foi escrito originalmente em português, o que é raro para um livro técnico nesta área. Segundo, ele é indicado para quem já possui alguma experiência em programação, mas que deseja uma noção prática do por que e de como usar Python para programar, sem ter que navegar por horas entre livros e manuais.

Uma observação: este tutorial não é um guia aprofundado, e nem um manual de referência. A documentação disponível em `http://www.python.org/docs/` é excelente (e melhor do que muitos dos livros publicados sobre Python), e serve de referência para os aspectos particulares da linguagem.

Convenções e Formato Este tutorial assume Python versão 2; a versão mais recente é a 2.3.3. O texto se baseia em um grande conjunto de exemplos, que aparecem indentados em uma seção distinta, em uma fonte de largura fixa. Os exemplos, de forma geral, descrevem precisamente a saída produzida ao digitar os comandos interativamente. Definições importantes são introduzidas em **negrito**. Nomes de variáveis e expressões de código vêm em uma fonte diferente, `com serifa`.

Quanto ao idioma, os termos em inglês mais comuns são utilizados, com uma explicação prévia do que significam. É importante que os termos em inglês sejam conhecidos, porque são utilizados correntemente na bibliografia e em discussões com outros programadores.

3.1.1 O que é Python?

Python é uma linguagem de programação[1]. Em outras palavras, e sendo simples ao extremo, usamos Python para escrever software. Esta linguagem tem alguns pontos que a tornam especial:

- É uma linguagem **interpretada**.
- Não há pré-declaração de variáveis, e os tipos das variáveis são determinados **dinamicamente**.

[1] Python é às vezes classificado como linguagem de *scripting*, um termo com o qual não concordo muito, já que Python é de uso geral, podendo ser usada para criar qualquer tipo de software. O termo *script* geralmente se refere a programas escritos em linguagens interpretadas que automatizam tarefas ou que 'conectam' programas distintos.

- O controle de bloco é feito apenas por **indentação**; não há delimitadores do tipo `BEGIN` e `END` ou { e }.
- Oferece **tipos de alto nível**: strings, listas, tuplas, dicionários, arquivos, classes.
- É **orientada a objetos**; aliás, em Python, tudo é um objeto.

Nas próximas seções estes aspectos são discutidos em detalhes.

Linguagem interpretada

Linguagens de programação são freqüentemente classificadas como compiladas ou interpretadas. Nas compiladas, o texto (ou **código-fonte**) do programa é lido por um programa chamado **compilador**, que cria um arquivo binário, executável diretamente pelo hardware da plataforma-alvo. Exemplos deste tipo de linguagem são C ou Fortran. A compilação e execução de um programa simples em C segue algo como:

```
% cc hello.c -o hello
% ./hello
Hello World
```

onde `cc` é o programa compilador, `hello.c` é o arquivo de código-fonte, e o arquivo criado, `hello`, é um executável binário.

Em contrapartida, programas escritos em linguagens interpretadas não são convertidos em um arquivo executável. Eles são executados utilizando um outro programa, o **interpretador**, que lê o código-fonte e o **interpreta** diretamente, durante a sua execução. Exemplos de linguagem interpretada incluem o BASIC tradicional, Perl e Python. Para executar um programa Python contido no arquivo `hello.py`, por exemplo, utiliza-se algo como:

```
% python hello.py
Hello World
```

Note que o programa que executamos diretamente é o interpretador `python`, fornecendo como parâmetro o arquivo com código-fonte `hello.py`. Não há o passo de geração de executável; o interpretador transforma o programa especificado à medida em que é executado.

Tipagem dinâmica

Um dos conceitos básicos em programação é a **variável**, que é uma associação entre um nome e um valor. Ou seja, abaixo, neste fragmento na linguagem C:

```
int a;
a = 1;
```

temos uma variável com o nome `a` sendo declarada, com tipo **inteiro** e contendo o valor 1. Em Python, não precisamos declarar variáveis, nem seus tipos:

```
>>> a = 1
```

seria a instrução equivalente; define uma variável com o valor 1, que é um valor inteiro.

Python possui o que é conhecido como **tipagem dinâmica**: o tipo ao qual a variável está associada pode variar durante a execução do programa. Não quer dizer que não exista tipo específico definido (a chamada **tipagem fraca**): embora em Python não o declaremos explicitamente, as variáveis sempre assumem um único tipo em um determinado momento.

```
>>> a = 1
>>> type(a)         # a função type() retorna o tipo
<type 'int'>        # associado a uma variável
>>> a = "1"
>>> type(a)
<type 'str'>
>>> a = 1.0
>>> type(a)
<type 'float'>
```

Tipagem dinâmica, além de reduzir a quantidade de planejamento prévio (e digitação!) para escrever um programa, é um mecanismo importante para garantir a simplicidade e flexibilidade das **funções** Python. Como os tipos dos argumentos não são explicitamente declarados, não há restrição sobre o que pode ser fornecido como parâmetro. No exemplo acima, são fornecidos argumentos de tipos diferentes à mesma função `type`, que retorna o tipo deste argumento.

Controle de bloco por indentação

Na maior parte das linguagens, há instruções ou símbolos específicos que delimitam blocos de código – os blocos que compõem o conteúdo de um laço ou expressão condicional, por exemplo. Em C:

```
if ( a < 0 ) {
    /* bloco de código */
}
```

ou em Fortran:

```
   if ( a .lt. 0 ) then
C       bloco de código
   endif
```

os blocos são delimitados explicitamente — em C por chaves, e em Fortran pelo par `then` e `endif`. Em Python, blocos de código são demarcados apenas por espaços formando uma indentação visual:

```
print "O valor de a é "
if a == 0:
    print "zero"
else:
    print a
```

Esta propriedade faz com que o código seja muito claro e legível — afinal, garante que a indentação esteja sempre correta — porém requer costume e um controle mais formal[2].

Tipos de alto nível

Além dos tipos básicos (inteiros, números de ponto flutuante, booleanos), alguns tipos pré-determinados em Python merecem atenção especial:

[2]Por exemplo, é importante convencionar se a indentação será feita por uma tabulação ou por um número determinado de espaços, já que todos editando um mesmo módulo Python devem usar o mesmo padrão.

Listas: como um vetor em outras linguagens, a lista é um conjunto (ou **seqüência**) de valores acessados (**indexados**) por um índice numérico, inteiro, começando em zero. A lista em Python pode armazenar valores de qualquer tipo.

```
>>> a = [  "A", "B", "C", 0, 1, 2 ]
>>> print a[0]
A
>>> print a[5]
2
```

Tuplas: tuplas são também seqüências de elementos arbitrários; se comportam como listas com a exceção de que são **imutáveis**: uma vez criadas não podem ser alteradas.

Strings: a cadeia de caracteres, uma forma de dado muito comum; a string Python á uma seqüência imutável, alocada dinamicamente, sem restrição de tamanho.

Dicionários: dicionários são seqüências que podem utilizar índices de tipos variados, bastando que estes índices sejam imutáveis (números, tuplas e strings, por exemplo). Dicionários são conhecidos em outras linguagens como arrays associativos ou *hashes*.

```
>>> autor = { "nome" : "Christian", "idade": 28 }
>>> print autor["nome"]
Christian
>>> print autor["idade"]
28
```

Arquivo: Python possui um tipo pré-definido para manipular arquivos; este tipo permite que seu conteúdo seja facilmente lido, alterado e escrito.

Classes e Instâncias: classes são estruturas especiais que servem para apoiar programação orientada a objetos; determinam um tipo customizado com dados e operações particulares. Instâncias são as expressões concretas destas classes. Orientação a objetos em Python é descrita em maiores detalhes na seção 3.4.

Orientação a objetos

Orientação a objetos (OO) é uma forma conceitual de estruturar um programa: ao invés de definirmos variáveis e criarmos funções que as manipulam, definimos **objetos** que possuem dados próprios e ações associadas. O programa orientado a objetos é resultado da 'colaboração' entre estes objetos.

Em Python, todos os dados podem ser considerados objetos: qualquer variável — mesmo as dos tipos básicos e pré-definidos — possui um valor e um conjunto de operações que pode ser realizado sobre este. Por exemplo, toda string em Python possui uma operação (ou **método**) chamada `upper`, que gera uma string nova com seu conteúdo em maiúsculas:

```
>>> a = "Hello"
>>> a.upper()
'HELLO'
```

Como a maior parte das linguagens que são consideradas 'orientadas a objeto', Python oferece um tipo especial para definir objetos customizados: a **classe**. Python suporta também funcionalidades comuns na orientação a objetos: herança, herança múltipla, polimorfismo, reflexão e introspecção.

3.1.2 Por que Python?

Dado que existe uma grande diversidade de linguagens diferentes, por que aprender Python é interessante ou mesmo importante? Na minha opinião, a linguagem combina um conjunto único de vantagens:

- Os conceitos fundamentais da linguagem são simples de entender.
- A sintaxe da linguagem é clara e fácil de aprender; o código produzido é normalmente curto e legível.
- Os tipos pré-definidos incluídos em Python são poderosos, e ainda assim simples de usar.
- A linguagem possui um interpretador de comandos interativo que permite aprender e testar rapidamente trechos de código.
- Python é expressivo, com abstrações de alto nível. Na grande maioria dos casos, um programa em Python será muito mais curto que seu correspondente escrito em outra linguagem. Isto também faz com o ciclo de desenvolvimento seja rápido e apresente potencial de defeitos reduzido – menos código, menos oportunidade para errar.
- Existe suporte para uma diversidade grande de bibliotecas externas. Ou seja, pode-se fazer em Python qualquer tipo de programa, mesmo que utilize gráficos, funções matemáticas complexas, ou uma determinada base de dados SQL.
- É possível escrever extensões a Python em C e C++ quando é necessário desempenho máximo, ou quando for desejável fazer interface com alguma ferramenta que possua biblioteca apenas nestas linguagens.
- Python permite que o programa execute inalterado em múltiplas plataformas; em outras palavras, a sua aplicação feita para Linux normalmente funcionará sem problemas em Windows e em outros sistemas onde existir um interpretador Python.
- Python é pouco punitivo: em geral, 'tudo pode' e há poucas restrições arbitrárias. Esta propriedade acaba por tornar prazeroso o aprendizado e uso da linguagem.
- Python é livre: além do interpretador ser distribuído como software livre (e portanto, gratuitamente), pode ser usado para criar qualquer tipo de software — proprietário ou livre. O projeto e implementação da linguagem é discutido aberta e diariamente em uma lista de correio eletrônico, e qualquer um é bem-vindo para propor alterações por meio de um processo simples e pouco burocrático.

Ao longo das próximas seções serão expostos aos poucos os pontos concretos que demonstram estas vantagens.

3.2 Python básico: invocação, tipos, operadores e estruturas

Esta primeira seção aborda os aspectos essenciais da linguagem. Por falta de um lugar melhor, será introduzido aqui o símbolo para comentários em Python: quando aparece o caracter sustenido (#)[3] no código-fonte, o restante da linha é ignorado.

[3]Também conhecido como 'número', 'jogo da velha', 'cardinal' e 'lasagna'.

3.2.1 Executando o interpretador Python interativamente

Python está disponível tanto para Windows e Macintosh como para os diversos Unix (incluindo o Linux). Cada plataforma possui um pacote específico para instalação, que normalmente inclui um grande número de bibliotecas de código e um programa executável `python`[4]. Este é o interpretador Python que usaremos para executar nossos programas.

Para iniciar, vamos executar o interpretador **interativamente**, no **modo shell**. Em outras palavras, vamos entrar o código-fonte diretamente, sem criar um arquivo separado. Para este fim, execute o interpretador (digitando apenas '`python`'). Será impresso algo como o seguinte (dependendo de qual sistema estiver usando):

```
% python
Python 2.1.1 (#5, Aug 23 2001, 01:38:34)
[GCC egcs-2.91.66 19990314/Linux] on linux2
Type "copyright", "credits" or "license" for information.
>>>
```

Este símbolo ■> chamamos de *prompt*, e indica que o interpretador está aguardando ser digitado um comando. O shell Python oferece uma excelente forma de testar código, e vamos começar com algumas instruções simples.

```
>>> a = 1
>>> print a
1
>>> b = "Hello world"
>>> print b
Hello world
```

Neste exemplo, são introduzidos dois aspectos básicos: primeiro, a atribuição de variáveis; segundo, a instrução `print`, que exibe valores e o conteúdo de variáveis.

A forma interativa de executar o Python é conveniente; no entanto, não armazena o código digitado, servindo apenas para testes e procedimentos simples. Para programas mais complexos, o código-fonte é normalmente escrito e armazenado em um arquivo.

3.2.2 Criando um programa e executando-o

Em Python, um arquivo contendo instruções da linguagem é chamado de **módulo**. Nos casos mais simples pode-se usar um único módulo, executando-o diretamente; no entanto, é interessante saber que é possível sub-dividir o programa em arquivos separados e facilmente integrar as funções definidas neles. Para criar um arquivo contendo um programa, basta usar qualquer editor de texto. Para um primeiro exemplo, crie um arquivo `hello.py` contendo as seguintes linhas:

```
a = "Hello"
b = "world"
print a,b
```

E a seguir, execute-o da seguinte forma:

```
% python hello.py
```

Ou seja, execute o interpretador e passe como parâmetro o nome do arquivo que contém o código a ser executado. O resultado impresso deve ser:

[4] Normalmente, é incluído um ambiente de edição e execução simples, como o IDLE, também escrito em Python. Estes podem ser também utilizados.

```
Hello world
```

Perceba bem que não há preâmbulo algum no código-fonte; escrevemos diretamente o código a ser executado[5].

Nas próximas seções do tutorial, estaremos utilizando bastante o modo shell; no entanto, para exercícios e exemplos mais extensos que algumas linhas, vale a pena usar módulos para permitir edição e revisão.

3.2.3 Tipos, variáveis e valores

Nomes de variáveis começam sempre com uma letra, não contém espaços, e assim como tudo em Python, são sensíveis a caixa (*case-sensitive*) — em outras palavras, minúsculas e maiúsculas fazem, sim, diferença. Como explicado anteriormente, a variável não precisa ser pré-declarada e seu tipo é determinado dinamicamente.

Tipos numéricos

Tipos numéricos representam valores numéricos. Em Python há alguns tipos numéricos pré-definidos: inteiros (**int**), números de ponto flutuante (**float**), booleanos (**bool**) e complexos (**complex**). Estes tipos suportam as operações matemáticas comuns como adição, subtração, multiplicação e divisão, e podem ser convertidos entre si.

> Uma observação: booleanos foram adicionados na versão Python 2.2; sua utilização é normalmente associada a expressões condicionais. Maiores detalhes sobre este tipo estão disponíveis na seção 3.2.4.

A seguir alguns exemplos de criação de variáveis numéricas:

```
>>> a = 1              # valor inteiro
>>> preco = 10.99      # valor ponto flutuante, ou float.
>>> t = True           # valor booleano
>>> i = 4+3j           # valor complexo
```

Valores inteiros podem também ser fornecidos em base octal e hexadecimal:

```
>>> a = 071
>>> print a
57
>>> a = 0xFF
>>> print a
255
```

Para ser considerado um **float**, o número deve possuir um ponto e uma casa decimal, mesmo que seja zero. O fato de ser considerado um float é importante para a operação divisão, pois dependendo do tipo dos operandos, a divisão é inteira ou em ponto flutuante.

```
>>> 5 / 2     # divisão inteira, resultado inteiro
2
>>> 5 / 2.0   # divisão em ponto flutuante
```

[5] Em Unix, é possível criar programas em Python e executá-los usando apenas o nome do módulo (sem digitar o nome do interpretador `python`). Basta adicionar a linha 'mágica' `#!/usr/bin/env python` ao início do módulo e dar permissão de execução ao arquivo. Note que isto não evita que seja necessário que o interpretador esteja instalado no sistema; apenas permite que se possa omitir digitar o seu nome.

```
2.5
>>> 5 * 2.13
10.649999999999999
```

Atenção especial para este terceiro exemplo, que demonstra uma particularidade da representação e impressão de números de ponto flutuante. Números deste tipo, como o valor `2.13` do exemplo, possuem uma representação interna particular devido à natureza dos intrínseca dos computadores digitais; operações realizadas sobre eles têm precisão limitada, e por este motivo o resultado impresso diverge ligeiramente do resultado aritmeticamente correto. Na prática, o resultado obtido é praticamente equivalente a 10.65[6].

Determinando o tipo de uma variável Para descobrir o tipo atual de uma variável, pode-se usar a função `type()`:

```
>>> a = 1
>>> print type(a)
<type 'int'>
>>> a = "hum"
>>> print type(a)
<type 'string'>
```

Uma das vantagens de Python são os tipos complexos que vêm pré-definidos, introduzidos na seção 3.1.1. As seções seguintes cobrem estes tipos.

Listas

A lista é uma **seqüência**: um conjunto linear (como um vetor em outras linguagens) de valores indexados por um número inteiro. Os índices são iniciados em **zero** e atribuídos seqüencialmente a partir deste. A lista pode conter quaisquer valores, incluindo valores de tipos mistos, e até outras listas. Para criar uma lista, usamos colchetes e vírgulas para enumerar os valores:

```
>>> numeros = [ 1, 2, 3 ]
>>> opcoes  = [ "nao", "sim", "talvez" ]
>>> modelos = [ 3.1, 3.11, 95, 98, 2000, "Millenium", "XP" ]
>>> listas  = [ numeros, opcoes ]
```

Neste exemplo criamos quatro listas diferentes, com elementos de tipos diversos. A quarta lista tem como elementos outras listas: em outras palavras, é uma lista de listas:

```
>>> print listas
[[1, 2, 3], ['nao', 'sim', 'talvez']]
```

Para acessar um elemento específico de uma lista, usamos o nome da lista seguido do índice entre colchetes:

```
>>> print numeros[0]
1
>>> print opcoes[2]
```

[6] Todas as linguagens de programação convencionais utilizam este mesmo mecanismo de representação, e na prática não resultam grandes problemas com isso, mas Python deixa transparecer esta particularidade em algumas situações, como esta. O tutorial Python em `http://www.python.org/docs/current/tut/` inclui uma seção sobre as nuances do uso e representação de números de ponto flutuante.

```
talvez
>>> print modelos[4]
2000
```

Usando índices negativos, as posições são acessadas a partir do final da lista, -1 indicando o último elemento:

```
>>> print numeros[-1]
3
>>> print modelos[-2]
Millenium
```

Python oferece um mecanismo para criar 'fatias' de uma lista, ou **slices**. Um slice é uma lista gerada a partir de um fragmento de outra lista. O fragmento é especificado com dois índices, separados por dois pontos; o slice resultante contém os elementos cujas posições vão do primeiro índice ao segundo, não incluindo o último elemento. Se omitirmos um dos índices no slice, assume-se início ou fim da lista.

```
>>> print numeros[:2]
[1, 2]
>>> print opcoes[1:]
['sim', 'talvez']
>>> print modelos[1:5]
[3.1000000000000001, 95, 98, 2000]
```

Note que o terceiro exemplo demonstra a mesma particularidade referente à representação de números de ponto flutuante, descrita na seção 3.2.3: ao imprimir a lista, Python exibe *a representação interna* dos seus elementos, e portanto, floats são impressos com precisão completa.

Métodos da Lista Na seção 3.1.1 introduzimos um conceito central da linguagem: 'tudo é um objetot'; uma lista em Python, sendo um objeto, possui um conjunto de operações próprias que manipulam seu conteúdo. O nome **método** é usado para descrever operações de um objeto; métodos são acessados no código-fonte digitando-se um ponto após o nome da variável, e a seguir o nome do método.

Para executar (ou **chamar**) um método, usamos parênteses após seu nome, fornecendo argumentos conforme necessário. Abaixo são exemplificados dois exemplos de chamadas de métodos em listas:

```
>>> numeros.append(0)
>>> print numeros
[1, 2, 3, 0]
>>> numeros.sort()
>>> print numeros
[0, 1, 2, 3]
```

Observe com atenção este exemplo, que demonstra os conceitos fundamentais descritos nos parágrafos anteriores:

- O método `append(v)` recebe como argumento um valor, e adiciona este valor ao final da lista.
- O método `sort()` ordena a lista, modificando-a. Não recebe argumentos.

A lista possui uma série de outros métodos; consulte a referência Python para uma descrição completa (e veja também a função `dir()` na seção 3.4.4).

Tuplas

A tupla é uma seqüência, como a lista: armazena um conjunto de elementos acessíveis por um índice inteiro. A tupla é imutável; uma vez criada, não pode ser modificada. Para criar uma tupla use parênteses, e vírgulas para separar seus elementos:

```
>>> t = ( 1, 3, 5, 7 )
>>> print t[2]
5
```

A tupla é utilizada em algumas situações importantes: como a lista de argumentos de uma função ou método, como chave em dicionários, e em outros casos onde fizer sentido ou for mais eficiente um conjunto fixo de valores.

Strings

A string, como citado anteriormente, é uma seqüência imutável com um propósito especial: armazenar cadeias de caracteres.

```
>>> a = "Mondo Bizarro"
>>> print a
Mondo Bizarro
```

Strings podem ser delimitadas tanto com aspas simples quanto duplas; se delimitamos com aspas duplas, podemos usar as aspas simples como parte literal da string, e vice-versa. Para inserir na string aspas literais do mesmo tipo que o delimitador escolhido, prefixe-as com uma contra-barra \. As atribuições abaixo são equivalentes:

```
>>> b = "All's quiet on the eastern front."
>>> c = 'All\'s quiet on the eastern front."
>>> b == c
True
```

São usados caracteres especiais para denotar quebra de linha (\n), tabulação (\t) e outros.

```
>>> a = "Hoje\n\t é o primeiro dia."
>>> print a
Hoje
    é o primeiro dia.
```

Para criar uma string com múltiplas linhas, é útil o delimitador aspas triplas: as linhas podem ser quebradas diretamente, e a string pode ser finalizada com outras três aspas consecutivas:

```
a = """I wear my sunglasses at night
So I can so I can
Keep track of the visions in my eyes"""
```

Finalmente, como toda seqüência, a string pode ser indexada ou dividida em slices, usando o operador colchetes:

```
>>> a = "Anticonstitucionalissimamente"
>>> print a[0]
A
```

```
>>> print a[13]
i
>>> print a[:4]
Anti
>>> print a[-5:-1]
ment
```

A string possui um operador especial, a porcentagem (`%`), que será descrito na seção 3.2.4. Possui ainda um grande número de métodos, descritos em detalhes na seção *String Methods* do manual de referência Python.

Dicionários

Um dicionário representa uma coleção de elementos onde é possível utilizar um índice de qualquer tipo imutável, ao contrário da lista, onde índices são sempre inteiros seqüencialmente atribuídos. É costumeiro usar os termos chave e valor (key/value) para descrever os elementos de um dicionário - a chave é o índice, e o valor, a informação correspondente àquela chave.

Para declarar dicionários, utilizamos o símbolo chaves, separando o índice do valor por dois pontos e separando os pares índice-valor por vírgulas:

```
>>> refeicoes = { "café" : "café", "almoço" : "macarrão",
...               "jantar" : "sopa" }
>>> print refeicoes["almoço"]
macarrao

>>> precos_modelos = { 98 : 89, 99 : 119, 2000 : 199 }
>>> print precos_modelos[98]
89
```

Neste exemplo criamos dois dicionários com três elementos cada um. As chaves do dicionário `refeicoes` são as strings ”café”, ”almoço” e ”jantar”, e os valores respectivos, as strings ”café”, ”macarrão” e ”sopa”.

Métodos do Dicionário O dicionário também possui alguns métodos notáveis:

- `keys()` retorna uma lista (sim, exatamente, do tipo lista) com as chaves do dicionário;
- `values()` retorna uma lista com os valores do dicionário;

```
>>> precos_modelos.keys()
[99, 98, 2000]
>>> precos_modelos.values()
[119, 89, 199]
    # A ordem dos elementos retornados por keys() e
    # values() é arbitrária; não confie nela.
```

- `has_key(k)` verifica se a lista possui aquela chave:

```
>>> precos_modelos.has_key(98)
True
>>> precos_modelos.has_key(97)
False
```

- `update(d2)` atualiza o dicionário com base em um segundo dicionário fornecido como parâmetro; elementos do dicionário original que também existem no segundo são atualizados; elementos que existem no segundo mas que não existem no original são adicionados a este.

```
>>> precos_modelos.update( { 2000 : 600, 2001: 700 } )
>>> print precos_modelos
{ 99: 400, 98: 300, 2001: 700, 2000: 600}
```

No exemplo acima, observe que a chave 2000 foi atualizada, e 2001, acrescentada.

É possível usar o dicionário como uma estrutura de dados simples, com campos para cada informação a ser armazenada. Por exemplo, para armazenar informação sobre um produto hipotético, com código, descrição e preço:

```
>>> produto = { "código":771, "desc":"Copo", "preço":10.22 }
>>> print produto["código"]
771
>>> print produto["desc"]
Copo
```

É um padrão comum criar listas de dicionários neste formato, cada ítem na lista correspondendo a um produto em particular.

3.2.4 Operadores

O próximo tópico essencial da linguagem são **operadores**, símbolos que operam sobre variáveis e valores.

Operadores aritméticos

A maior parte dos operadores aritméticos em Python funciona de maneira intuitiva e análoga aos operadores em outras linguagens. Demonstrando por exemplo:

```
>>> print a + 3       # adição
10
>>> print a - 2       # subtração
5
>>> print a / 2       # divisão inteira: argumentos inteiros
3                     # e resultado inteiro

>>> print a / 2.5     # divisão em ponto flutuante: pelo
2.8                   # menos um argumento deve ser float

>>> print a % 4       # resto da divisão inteira
3
>>> print a * 2       # multiplicação
14
>>> print a ** 2      # exponenciação
49
```

A exponenciação também pode ser feita por meio de uma função, `pow()`, como descrito na seção 3.3. A raiz quadrada e outras funções matemáticas estão implementadas no módulo `math`; veja a seção 3.5 para maiores detalhes.

Com exceção da exponenciação e da divisão inteira, estes operadores são bastante comuns em linguagens de programação. Os operadores aritméticos podem ser usados em floats também:

```
>>> a = 1.15
>>> print a / a - a * a + a
0.57349375
```

e os operadores de adição (+) e multiplicação (*), em strings:

```
>>> a = "exato"
>>> print a * 2
exatoexato
>>> print "quase " + a
quase exato
```

, listas:

```
>>> a = [ -1, 0 ]
>>> b = [ 1, 2, 3 ]
>>> print b * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> print a + b
[1, 2, 3, 1, 2]
```

e tuplas:

```
>>> a = ( 1, 2 )
>>> print a + (2, 1)
(1, 2, 2, 1)
```

Como exemplificado acima, o operador adição (e multiplicação) serve para concatenar listas, tuplas e strings. Não pode ser utilizado com dicionários (que podem ser atualizados usando a função `update()`, mas para os quais a operação de concatenação não faria sentido).

Operadores sobre cadeias de bits

Para inteiros, existem operadores que manipulam seus valores como cadeias de bits (operadores *bit-wise*) de acordo com a aritmética booleana:

```
>>> a = 0xa1
>>> b = 0x01
>>> print a, b  # para imprimir os valores decimais
161, 1
>>> a & b       # and
1
>>> a | b       # or
161
>>> a << 1      # shift para esquerda
322
>>> b >> 1      # shift para direita
0
>>> ~a          # inversão em complemento de 2
-162
```

Note que a representação dos valores inteiros, por padrão, é decimal: aqui, embora fornecemos os valores em base hexadecimal, os resultados aparecem na base 10.

Operadores de atribuição

O operador mais simples, e que já fui utilizado em diversos exemplos anteriores, é o operador de atribuição. Este operador é representado por um único símbolo de igualdade, `=`, definindo uma variável e automaticamente atribuindo a ela um valor. O exemplo abaixo define uma variável a, com valor inteiro 1.

```
>>> a = 1
```

Existem formas variantes deste operador que o combinam com os operadores aritméticos e bit-wise introduzidos nas seções anteriores. Estas formas foram introduzidos na versão Python 2.0, e existem primariamente para oferecer uma maneira conveniente de re-atribuir um valor transformado a uma variável.

A sintaxe para o operador combinado utiliza o símbolo do operador aritmético/bit-wise relevante, seguido da igualdade. Este operador combinado efetua a operação sobre o valor da variável, já atribuindo o resultado a esta mesma. Exemplificando:

```
>>> a = 1
>>> a += 1
>>> print a
2
>>> a /= 2
>>> print a
1
>>> a *= 10
>>> print a
10
```

Operadores condicionais

Tradicionalmente, programação envolve testar valores (e tomar decisões com base no resultado do teste). Um teste é essencialmente uma expressão condicional que tem um resultado verdadeiro ou falso. Esta seção descreve os operadores condicionais mais comuns em Python.

> A partir da versão 2.2 de Python, existe um tipo numérico que representa o resultado de um teste condicional: o tipo booleano. É um tipo bastante simples, possuindo apenas dois valores possíveis: `True` e `False`. A partir da versão 2.3 de Python, qualquer comparação retorna um valor booleano; versões anteriores retornavam 1 ou 0, respectivamente. Na prática, o resultado é equivalente, ficando apenas mais explícito e legível o resultado.

Igualdade

```
>>> print 2 == 4            # igualdade
False
>>> print 2 != 4            # diferente de
True
>>> print "a" == "a"
True
>>> print "a" != "b"
True
```

Comparação

```
>>> print 1 < 2            # menor que
True
>>> print 3 > 5            # maior que
False
>>> print 3 >= 4           # maior ou igual que
False
```

Comparações podem ser realizadas também com outros tipos de variáveis; para strings, listas ou dicionários, as comparações 'maior que' e 'menor que' se referem ao número e ao valor dos elementos, mas por não serem terrivelmente claras não são muito freqüentemente usadas[7].

```
>>> [ 2,2,3 ] < [ 2,3,2 ]
True
>>> "abacate" < "pera"
True
```

> **Dica:** quando comparando o número de elementos de uma seqüência ou dicionário, é normalmente utilizada a função `len()`, que retorna o 'comprimento' da seqüência.

Presença em Seqüências Para seqüências e dicionários, existe o operador `in`, que verifica se o elemento está contido no conjunto; no caso do dicionário, o operador testa a presença do valor na seqüência de chaves.

```
>>> "x" in "cha"
False
>>> 1 in [ 1,2,3 ]
True
```

Operadores lógicos

Os operadores lógicos `not`, `and` e `or` permitem modificar e agrupar o resultado de testes condicionais:

```
>>> nome = "pedro"
>>> idade = 24
>>> nome == "pedro" and idade == 25
False
>>> nome == "pedro" and idade < 25
True
>>> len(nome) < 10 or not nome == "pedro"
False
```

Estes operadores são utilizados com freqüência nas estruturas de controle descritas na seção 3.2.5.

Combinação de operadores Python oferece uma forma implícita de combinar operações condicionais, sem o uso de operadores lógicos. Por exemplo, para verificar se um valor está entre dois extremos, pode-se usar a seguinte sintaxe:

[7] A exceção óbvia é na ordenação de seqüências contendo elementos destes tipos.

```
if 0 < a < 10:
    print "Entre zero e dez"
```

Podem também ser comparados diversos valores simultaneamente:

```
if a == b == c:
    print "São idênticos"
```

e mesmo combinados operadores comparativos com operadores de igualdade:

```
if a == b <= c:
    print "São idênticos"
```

Esta expressão verifica se `a` é igual a `b` e além disso, se `b` é menor ou igual a `c`.

Substituição em strings: o operador %

Uma operação muito útil para processamento de texto é a substituição de símbolos em strings. É particularmente adequada para gerarmos strings formatadas contendo algum valor variável, como o clássico formulário: "Nome: _____ Idade: _____ anos".

1. Escreve-se a string normalmente, usando um símbolo especial no lugar da lacuna:

   - `%d`: para substituir inteiros
   - `%f`: para substituir floats
   - `%s`: para substituir outra string

   ```
   >>> a = "Total de itens: %d"
   >>> b = "Custo: %f"
   ```

2. Para efetuar a substituição, aplica-se um operador `%` sobre a string contendo o símbolo de formatação, seguido do valor ou variável a substituir:

   ```
   >>> print a % 10
   Total de itens: 10
   ```

   Como pode ser visto, o símbolo é substituído pelo valor fornecido. Podem ser utilizados tanto valores explícitos quanto variáveis para a substituição:

   ```
   >>> custo = 5.50
   >>> print b % custo
   Custo: 5.500000
   ```

   Caso sejam múltiplos valores a substituir, use uma tupla:

   ```
   >>> print "Cliente: %s, Valor %f" % ( "hungry.com", 40.30 )
   Fornecedor: hungry.com, Custo 40.300000
   ```

Este operador permite ainda utilizar um número junto ao símbolo porcentagem para reservar um tamanho total à string:

```
>>> a = "Quantidade: %4d"
>>> print a % 3
>>> print a % 53
>>> print a % 120
Quantidade:    3
Quantidade:   53
Quantidade:  120
```

É possível controlar a formatação de tipos numéricos de maneira especial através de modificadores nos símbolos no formato `m.n`. Como acima, `m` indica o total de caracteres reservados. Para floats, `n` indica o número de casas decimais; para inteiros, indica o tamanho total do número, preenchido com zeros à esquerda. Ambos os modificadores podem ser omitidos.

```
>>> e = 2.7313
>>> p = 3.1415
>>> sete = 7
>>> print "Euler: %.7f" % e        # 7 casas decimais
Euler: 2.7313000
>>> print "Pi: %10.3f" % p         # 10 espaços, 3 casas
Pi:      3.142
>>> print "Sete: %10.3d" % sete    # 10 espaços, 3 dígitos
Sete:        007                  # (é um inteiro)
```

3.2.5 Estruturas de controle

Toda linguagem de programação possui instruções que controlam o fluxo de execução; em Python, há um conjunto pequeno e poderoso de instruções, descritas nas seções a seguir.

Condicional: a instrução `if`

A instrução condicional básica de Python é o `if`. A sintaxe é descrita a seguir (lembrando que a indentação é que delimita o bloco):

```
if condição:
    # bloco de código
elif condição:
    # outro bloco
else:
    # bloco final
```

As condições acima são comparações feitas utilizando os operadores condicionais descritos na seção 3.2.4, possivelmente combinados por meio dos operadores lógicos descritos na seção 3.2.4.

```
a = 2
b = 12
if a < 5 and b * a > 0:
    print "ok"
```

A instrução elif permite que se inclua uma exceção condicional — algo como "... **senão** se isso ...". O else é uma exceção absoluta[8].

[8] Não há nenhuma estrutura do tipo `switch/case`, mas é possível simulá-la usando uma expressão `if` com um `elif` para cada caso diferente.

```
if nome == "pedro":
    idade = 21
elif nome == "josé":
    idade = 83
else:
    idade = 0
```

Provendo `nome="álvaro"` para o bloco acima, será atribuído o valor zero a `idade`.

Laço iterativo: `for`

Há apenas dois tipos de laços em Python: `for` e `while`. O primeiro tipo, mais freqüentemente utilizado, percorre uma seqüência em ordem, a cada ciclo substituindo a variável especificada por um dos elementos. Por exemplo:

```
>>> jan_ken_pon = [ "pedra", "papel", "cenoura" ]
>>> for item in jan_ken_pon:
...   print item
...
pedra
papel
cenoura
```

A cada iteração, `item` recebe o valor de um elemento da seqüência. Para efetuar uma laço com um número fixo de iterações, costuma-se usar o `for` em conjunto com a função `range`, que gera seqüências de números:

```
>>> for i in range(1,4):
...   print "%da volta" % i
...
1a volta
2a volta
3a volta
```

Iteração em dicionários Para iterar em dicionários, podemos usar as funções `keys()` ou `values()` para gerar uma lista:

```
>>> dict = { "batata" : 500, "abóbora" : 1200,
...          "cebola" : 800 }
>>> for e in dict.keys():
...   print "Item: %8s Peso: %8s" % ( e, dict[e] )
Item: cebola  Peso:   800
Item: batata  Peso:   500
Item: abóbora Peso:  1200
```

Note que porque o dicionário em si não possui um conceito de ordem, a lista que resulta do método `keys()` possui ordenação arbitrária. Por este motivo, no exemplo acima, o laço não segue a declaração do dicionário.

Controle adicional em laços Para ambos os tipos de laço, existem duas instruções de controle adicional, `break` e `continue`. A primeira reinicia uma nova iteração imediatamente, interrompendo a iteração atual; a segunda faz o laço terminar imediatamente.

A forma geral do laço `for` é:

```
for variável in seqüência:
    # bloco de código
else:
    # bloco executado na ausência de um break
```

Note a presença da cláusula `else`. Esta cláusula é executada quando a saída do laço *não for determinada* por uma instrução `break`. Um exemplo clarifica este mecanismo:

```
valores = [2, 4, 5, 2, -1]
for i in valores:
    if i < 0:
        print "Negativo encontrado: %d" % i
        break
else:
    print "Nenhum negativo encontrado"
```

Laço condicional: `while`

O segundo tipo de laço, `while`, é utilizado quando necessitamos fazer um teste a cada iteração do laço.

```
while condição:
    # bloco de código
else:
    # bloco executado na ausência de um break
```

Como o laço for, o while possui uma cláusula `else`. Um exemplo do uso de while segue:

```
>>> m = 3 * 19
>>> n = 5 * 13
>>> contador = 0
>>> while m < n:
...   m = n / 0.5
...   n = m / 0.5
...   contador = contador + 1
...
>>> print "Iteramos %d vezes." % contador
Iteramos 510 vezes.
```

Não há uma instrução especial para efetuar um laço com teste ao final da iteração (como o laço `do ... while() em C`), mas pode-se usar um while infinito — usando uma condição verdadeira, fixa — em combinação com um teste e `break` internos:

```
>>> l = [ "a", "b", "c" ]
>>> i = len(l)
>>> while True:
...   if i < 0:
...     break
```

```
...   print l[i]
...   i = i - 1
...
c
b
a
```

Veracidade e falsidade de condições As estruturas `if` e `while` utilizam condições lógicas para controle, avaliando-as de maneira booleana. Como qualquer valor ou expressão pode ser usado como condição, é importante entender qual o mecanismo de avaliação da linguagem.

Em Python, falso é denotado:

- pelo booleano False,
- pelo valor 0 (zero),
- pela lista, dicionário, tupla ou string vazios, de tamanho zero,
- pelo valor especial None, que significa nulo.

Qualquer outro valor simples é considerado verdadeiro. Instâncias podem definir regras mais complexas para avaliação; para detalhes consulte a seção *Basic customization* do manual de referência Python.

Exceções

Com os dois tipos de laços descritos na seção anterior, todas as necessidades 'normais' de controle de um programa podem ser implementadas. No entanto, quando algo inesperado ocorre, ou uma condição de erro conhecido é atingida, Python oferece uma forma adicional de controlar o fluxo de execução: a exceção.

A exceção é um recurso de linguagens de programação modernas que serve para informar que uma condição incomum ocorreu. Embora existam outras aplicações, em geral comunicam-se através de exceções erros ou problemas que ocorrem durante a execução de um programa.

Exceções são internamente geradas pelo interpretador Python quando situações excepcionais ocorrem. No exemplo abaixo,

```
>>> a = [ 1, 2, 3 ]
>>> print a[5]
```

o código interno do interpretador sabe que não podemos acessar uma lista através um índice não-existente, e gera uma exceção:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
```

Vamos analisar a mensagem exibida. A primeira linha anuncia um **traceback**, que é a forma como é exibida a pilha de execução que gerou a exceção. A segunda indica a linha de código na qual ocorreu o problema, e o arquivo. Como estamos usando o modo interativo neste exemplo, o arquivo aparece como `<stdin>`, que é a entrada padrão. A terceira linha indica o tipo de exceção levantada — neste caso, `IndexError` — e informa uma mensagem que descreve mais especificamente o problema.

Tratando exceções A exceção normalmente imprime um traceback e interrompe a execução do programa. Uma ação comum é testar e controlar a exceção; para este fim, usamos uma cláusula `try/except`:

```
>>> a = [ 1, 2, 3 ]
>>> try:
...   print a[5]
... except IndexError:
...   print "O vetor nao possui tantas posições!"
O vetor nao possui tantas posições!
```

A instrução `try` captura exceções que ocorrerem no seu bloco de código; a linha `except` determina quais tipos de exceção serão capturados. A sintaxe da cláusula except segue os formatos:

```
except tipo_da_excecao [ , variavel_da_excecao ]
    # bloco de código
```

ou

```
except ( tipo_excecao_1, tipo_excecao_2, ... )
       [ , variavel_da_excecao ]
    # bloco de código
```

O primeiro elemento da cláusula `except` é um tipo da exceção, ou uma tupla de tipos caso múltiplas exceções devam ser tratadas da mesma forma. O segundo elemento é opcional; permite que seja capturada uma instância que armazena informações da exceção. Um uso trivial desta instância é imprimir a mensagem de erro:

```
>>> a = "foo"
>>> print a + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot add type "int" to string
```

Podemos capturar e tratar de forma especial o erro acima, imprimindo a mensagem fornecida e continuando a execução normalmente.

```
>>> try:
...   print a + 1
... except TypeError, e:
...   print "Um erro ocorreu: %s" % e
...
Um erro ocorreu: cannot add type "int" to string
>>>
```

Diversos tipos de exceções vêm pré-definidas pelo interpretador Python; o guia de referência contém uma lista completa e os casos onde são levantadas. Note também que esta introdução a exceções não cobre a sua sintaxe completa; consulte a seção *Errors and Exceptions* do tutorial Python para maiores detalhes.

3.2.6 Funções

Funções são blocos de código com um nome; recebem um conjunto de parâmetros (ou **argumentos**) e retornam um valor. Python possui, como seria esperado de uma linguagem de programação completa, suporte a funções. Existem diversas funções pré-definidas pelo interpretador, descritas na seção 3.3; a seção atual detalha as formas de definir funções em código Python.

Sintaxe básica

A sintaxe geral para definir uma função é:

```
def nome_funcao ( arg_1, arg_2, ..., arg_n ):
    #
    # bloco de código contendo corpo da função
    #
    return valor_de_retorno # retornar é opcional
```

O código da função, uma vez avaliado pelo interpretador, define um nome local. Os argumentos são valores fornecidos quando chamada a função, e que ficam disponíveis por meio de variáveis locais no corpo da função.

No exemplo abaixo, temos uma função que recebe uma lista de dicionários como parâmetro, e uma função que recebe um dicionário. A primeira itera pela lista recebida, e passa o elemento atual como argumento para a segunda.

```
def imprime_cardapio (pratos):
    print "Cardapio para hoje\n"
    for p in pratos:
        imprime_prato(p)
    print "\nTotal de pratos: %d" % len(pratos)

def imprime_prato(p)
  print "%s ........ %10.2f" % (p["nome"], p["preco"])
```

Ao ser interpretado, o código acima define dois nomes locais: `imprime_cardapio` e `imprime_prato`. Para que a função seja de fato executada, usamos este nome seguido dos argumentos passados entre parênteses:

```
# defino dicionários descrevendo os pratos
p1 = {"nome": "Arroz com brocolis", "preco": 9.90}
p2 = {"nome": "Soja com legumes",   "preco": 7.80}
p3 = {"nome": "Lentilhas", "preco": 4.80 }
lista_pratos = [p1, p2, p3]

# e chamo uma função, passando os pratos como argumento
imprime_cardapio(lista_pratos)
```

o que resulta na seguinte saída quando executado:

```
Cardapio para hoje

    Arroz com brocolis ........... 9.90
      Soja com legumes ........... 7.80
             Lentilhas ........... 4.80

Total de pratos: 3
```

Retornando um valor No exemplo acima, as funções não 'retornam' valor algum, apenas exibindo informação. Para retornar um valor, basta usar a expressão `return` dentro da função. A primeira pergunta conceitual aqui seria 'retornar para quem?'; a resposta direta é 'para quem invocou a função'. Demonstrando por exemplos, a função abaixo:

```
def bar(t):
    return "The rain in Spain falls mainly in the %s." % t
```

define um valor de retorno, que é uma string; ao chamar esta função com um argumento:

```
a = bar("plains")
```

o código da função é avaliado e um valor, retornado, sendo armazenado na variável `a` (observe o uso do operador de atribuição seguido da chamada da função). Para visualizar o valor retornado, usamos a conveniente instrução `print`:

```
>>> print a
The rain in Spain falls mainly in the plains.
```

Dica: para retornar conjuntos de valores, basta retornar uma seqüência ou outro valor de tipo composto.

Truques com argumentos

Argumentos nomeados Argumentos podem ser fornecidos à função especificando-se o nome do parâmetro ao qual correspondem. O código abaixo é equivalente à chamada no exemplo anterior:

```
a = bar(t="plains")
```

mas fica explicitamente declarado que o parâmetro corresponde ao argumento t. Normalmente são nomeados argumentos para tornar a leitura do código mais fácil; no entanto, veremos a seguir como são essenciais para utilizar uma das formas de listas de argumentos variáveis.

Valores padrão Uma das funcionalidades mais interessantes nas funções Python é a possibilidade de definir valores padrão:

```
def aplica_multa(valor, taxa=0.1):
    return valor + valor * taxa
```

Neste exemplo, o valor padrão para a variável `taxa` é 0.1; se não for definido este argumento, o valor padrão será utilizado.

```
>>> print aplica_multa(10)
11.0
>>> print aplica_multa(10, 0.5)
15.0
```

Dica: Não utilize como valor padrão listas, dicionários e outros valores mutáveis; os valores padrão são avaliados apenas uma vez e o resultado obtido não é o que intuitivamente se esperaria.

Conjuntos de argumentos opcionais Uma forma de definir conjuntos de argumentos opcionais utiliza parâmetros 'curinga' que assumem valores compostos. Pode ser fornecido um parâmetro cujo nome é prefixado por um símbolo asterisco, que assumirá um conjunto ordenado de argumentos:

```
def desculpa(alvo, *motivos):
    d = "Desculpas %s, mas estou doente" % alvo
    for motivo in motivos:
        d = d + " e %s" % motivo
    return d + "."

>>> desculpa("senhor", "meu gato fugiu",
...          "minha tia veio visitar")

"Desculpas senhor, mas estou doente e meu gato fugiu e minha
tia veio visitar."
```

ou um parâmetro que assume um conjunto de argumentos nomeados, prefixado de dois asteriscos:

```
def equipe(diretor, produtor, **atores):
    for personagem in atores.keys():
        print "%s: %s" % (personagem, atores[personagem])
    print "-" * 20
    print "Diretor:  %s" % diretor
    print "Produtor: %s" % produtor

>>> equipe(diretor="Paul Anderson",
...        produtor="Paul Anderson",
...        Frank="Tom Cruise", Edmund="Pat Healy",
...        Linda="Julianne Moore")

Frank: Tom Cruise
Edmund: Pat Healy
Linda: Julianne Moore
--------------------
Diretor: Paul Anderson
Produtor: Paul Anderson
```

ou ainda as duas formas combinadas. Estes parâmetros especiais devem necessariamente ser os últimos definidos na lista de parâmetros da função. Note que estes argumentos especiais não agregam parâmetros que correspondem a argumentos explicitamente definidos, como `alvo`, `diretor` e `produtor` nos exemplos acima; estes são processados de maneira normal.

Conjuntos de argumentos opcionais são uma excelente forma de conferir flexibilidade a uma função mantendo compatibilidade com código pré-existente, sem prejudicar a sua legibilidade.

Escopo de variáveis Embora o tema escopo seja complexo, e não esteja restrito apenas a funções, cabe aqui uma observação sobre o escopo das variáveis definidas no corpo de funções.

Tanto as variáveis definidas no corpo de uma função, como a variável `i` no exemplo abaixo, quanto os argumentos da função são acessíveis apenas no **escopo local**; em outras palavras, apenas no bloco de código da própria função.

```
v = 0
w = 1
def verifica(a, b):
    i = 0
    # Neste bloco de código, v, w, a, b e i são acessíveis.
```

```
def cancela(x, y):
    i = 0
    # Neste bloco de código, v, w, x, y e i são acessíveis,
    # mas note que *este* i não tem nenhuma relação com
    # o i da função verifica() acima.
```

Variáveis definidas no corpo principal (ou seja, definidas em um bloco não-indentado, como as variáveis v e w acima) de um módulo podem ser *lidas* em qualquer função contida naquele arquivo; no entanto, não podem ser alteradas[9].

Caso se deseje definir ou atribuir um novo valor a uma variável global, existe uma instrução especial, `global`. Esta instrução indica que a variável cujo nome for especificado é para ser definida no escopo global, e não local.

```
v = 0
def processa(t):
    global v
    v = 1
```

No código acima, a função altera o valor da variável global v.

Há duas funções úteis para estudar os escopos que se aplicam em um determinado contexto:

- `locals()`, que retorna um dicionário descrevendo o escopo local ao contexto atual; os itens do dicionário são compostos dos nomes das variáveis definidas neste escopo e os valores aos quais correspondem.
- `global()`, que retorna um dicionário semelhante ao da função `locals()`, mas que descreve o escopo global.

O escopo de variáveis funciona de maneira semelhante quando tratando com métodos e definições de classes, que serão discutidos na seção 3.4.

3.2.7 Módulos e o comando `import`

Como dito anteriormente, cada arquivo contendo código Python é denominado um **módulo**. Na grande maioria das ocasiões utilizamos um ou mais módulos Python em combinação: o interpretador interativo é adequado para realizar experimentos curtos, mas não para escrever código de produção.

Um módulo Python consiste de código-fonte contido em um arquivo denominado com a extensão `.py`; como tal, pode conter variáveis, funções e classes; para fins de nomenclatura, qualquer um destes elementos contidos em um módulo é considerado um **atributo do módulo**.

Python, através do módulo, oferece excelentes mecanismos para modularizar código-fonte. Esta modularização pode ter diversas motivações: o programa pode ser grande demais, ter sub-partes reusáveis que devem ser separadas, ou ainda necessitar de módulos escritos por terceiros. Esta seção introduz este conceito através do comando `import`.

Importando módulos e atributos de módulos A instrução básica para manipular módulos é `import`. O módulo deve estar no caminho de procura de módulos do interpretador[10]. No exemplo a seguir, um módulo com o nome `os.py` é importado. Observe que a extensão `.py` *não* é incluída no comando `import` — apenas o nome principal:

[9] O motivo pelo qual existe esta distinção está intimamente associado à forma como Python atribui variáveis. Ao realizar uma operação de atribuição, a variável é sempre definida no escopo *local*; acesso à variável, no entanto, efetua uma busca nos escopos, do mais local para o mais global.

[10] O caminho de procura é uma lista de diretórios onde o interpretador Python busca um módulo quando uma instrução `import` é processada. Pode ser manipulado pela variável de ambiente `PYTHONPATH` ou pelo módulo `sys.path`.

```
>>> import os
>>> print os.getcwd()
/home/kiko
```

O módulo `os` define algumas funções internamente. No exemplo acima, invocamos a função `getcwd()` contida neste, *prefixando a função com o nome do módulo*. É importante esta observação: ao usar a forma `import módulo`, acessamos os atributos de um módulo usando esta sintaxe, idêntica à utilizada para acessar métodos da lista conforme descrito na seção 3.2.3.

Existe uma segunda forma do comando `import` que funciona de forma diferente. Ao invés de importar o módulo inteiro, o que nos obriga a usar as funções prefixadas pelo nome do módulo, este formato importa um atributo do módulo, deixando-o acessível localmente:

```
>>> from os import getcwd
>>> print getcwd()
/home/kiko
```

Funções úteis Há algumas funções pré-definidas no interpretador bastante úteis quando lidando com módulos e os atributos contidos em módulos:

- `dir(nome_módulo)` retorna uma lista dos nomes dos atributos contidos em um módulo, o que permite que você descubra interativamente quais símbolos e funções o compõem. Experimente, por exemplo, executar `print dir(os)` a partir do interpretador Python.
- `reload(nome_módulo)` recarrega o módulo importado a partir do seu arquivo; desta maneira, alterações podem ser efetuadas no arquivo do módulo e já utilizadas em uma sessão do interpretador, sem que seja necessário reiniciá-lo.

O mecanismo de importação possui uma série de nuances especiais, que estão associados ao tópico escopo, introduzido na seção anterior, e aos *namespaces*, que resumidamente determinam o conjunto de atributos acessíveis em um determinado contexto. Uma descrição mais detalhada destes tópicos é oferecida na seção *Python Scopes and Name Spaces* do tutorial Python.

3.2.8 Strings de documentação

Um recurso especialmente útil em Python é o suporte nativo à documentação de código por meio de strings localizadas estrategicamente, chamadas **docstrings**. Módulos, classes, funções e até propriedades podem ser descritas por meio de docstrings; o exemplo a seguir demonstra um módulo hipotético, `financ.py`, documentado adequadamente:

```
"""Módulo que contém funções financeiras."""

def calcula_juros(valor, taxa=0.1):
    """Calcula juros sobre um valor.

    Aplica uma taxa de juros fornecida sobre um valor
    e retorna o resultado. Se omitida a taxa, o valor
    0.1 será utilizado"""
    # ...

class Pagamento:
    """Classe que representa um pagamento a ser efetuado.

    Inclui informações de crédito e débito. Permite efetuar
```

```
    operações como devolução, cancelamento, transferência e
    pagamento em si. Possui o seguinte ciclo de vida:
    ...
    """
```

As docstrings do módulo acima podem ser visualizadas em tempo de execução, e mesmo a partir do modo interativo do interpretador por meio da função `help()` e do atributo `__doc__`:

```
>>> import financ
>>> help(calcula_juros)
Calcula juros sobre um valor.

    Aplica uma taxa de juros fornecida sobre um valor
    e retorna o resultado. Se omitida a taxa, o valor
    0.1 será utilizado

>>> print financ.__doc__
Módulo que contém funções financeiras.
```

Observe que a função `help` recebe como argumento *a própria função* `calcula_juros`; pode parecer pouco usual mas é um padrão comum em Python.

Este recurso é extremamente útil para o aprendizado da linguagem quando explorando objetos pré-definidos; utilize-o sempre que estiver necessitando de uma referência rápida:

```
>>> len([1,2,3])
3
>>> help(len)
Help on built-in function len:

len(...)
    len(object) -> integer

    Return the number of items of a sequence or mapping.
```

Docstrings podem também ser processadas utilizando ferramentas externas, como o **epydoc**[11], gerando referências em formatos navegáveis como HTML.

3.3 Funções pré-definidas

Python possui uma série de funções pré-definidas, que já estão disponíveis quando executamos o interpretador, sem ter que recorrer a bibliotecas externas. Algumas funções importantes que ainda não foram apresentadas no texto seguem:

- `range(a,b)`: recebe dois inteiros, retorna uma lista de inteiros entre `a` e `b`, não incluindo `b`. Esta função é freqüentemente utilizada para iterar laços `for`.

  ```
  >>> print range(1,10)
  [1, 2, 3, 4, 5, 6, 7, 8, 9]
  ```

- `len(a)`: retorna o comprimento da variável `a`: para listas, tuplas e dicionários, retorna o número de elementos; para strings, o número de caracteres; e assim por diante.

[11] Disponível em `http://epydoc.sourceforge.net/`; um exemplo da documentação gerada pode ser consultado em `http://www.async.com.br/projects/kiwi/api/`.

```
>>> print len([1,2,3])
3
```

- `round(a, n)`: recebe um float e um número; retorna o float arredondado com este número de casas decimais.
- `pow(a, n)`: recebe dois inteiros; retorna o resultado da exponenciação de `a` à ordem `n`. É equivalente à sintaxe `a ** n`.
- `chr(a)`: recebe um inteiro (entre 0 e 255) como parâmetro, retornando o caracter correspondente da tabela ASCII.

```
>>> print chr(97)
"a"
```

- `unichr(a)`: como `chr()`, recebe um inteiro (aqui variando entre 0 e 65535), retornando o caracter Unicode correspondente.
- `ord(a)`: recebe um único caracter como parâmetro, retornando o seu código ASCII.

```
>>> print ord("a")
97
```

- `min(a, b)`: retorna o menor entre `a` e `b`, sendo aplicável a valores de qualquer tipo.
- `max(a, b)`: retorna o maior entre `a` e `b`.
- `abs(n)`: retorna o valor absoluto de um número.
- `hex(n)` e `oct(n)`: retornam uma string contendo a representação em hexadecimal e octal, respectivamente, de um inteiro.

Há também funções de conversão explícita de tipo; as mais freqüentemente utilizadas incluem:

- `float(n)`: converte um inteiro em um float.

```
>>> print float(1)
1.0
```

- `int(n)`: converte um float em inteiro.

```
>>> print int(5.5)
5
```

- `str(n)`: converte qualquer tipo em uma string. Tipos seqüencias são convertidos de forma literal, peculiarmente.

```
>>> print str([1,2,3]), str({"a": 1})
[1, 2, 3] {'a': 1}
```

- `list(l)` e `tuple(l)`: convertem uma seqüencia em uma lista ou tupla, respectivamente.

```
>>> print list("ábaco")
['á', 'b', 'a', 'c', 'o']
```

Além destas funções, existe uma grande biblioteca disponível nos módulos já fornecidos com o Python. Alguns destes módulos são discutidos na seção 3.5; como sempre, o manual Python é a referência definitiva no assunto.

As seções seguintes discutem algumas funções pré-definidas com comportamento especialmente relevante.

3.3.1 Manipulação de arquivos: a função `open()`

A função `open` é uma das mais poderosas do Python; serve para obter uma referência a um objeto do tipo arquivo. Assumindo que temos um arquivo chamado `arquivo.txt`, contendo um trecho de um livro famoso, podemos codificar o seguinte exemplo:

```
>>> a = open("arquivo.txt")
>>> print a
<open file 'arquivo.txt', mode 'r' at 0x820b8c8>
```

Uma vez obtida a referência ao objeto arquivo, podemos usar métodos específicos deste, como o método `read()`, que retorna o conteúdo do arquivo:

```
>>> texto = a.read()
>>> print texto
'...Would you tell me, please,
 which way I ought to go from here?'
'That depends a good deal on where you want to get to,'
 said the Cat.
'I don't much care where--' said Alice.
'Then it doesn't matter which way you go,' said the Cat.
```

Sintaxe O formato geral da função `open` é:

```
open( nome_do_arquivo, modo )
```

Ambos os parâmetros são strings. O modo determina a forma como o arquivo será aberto e é composto de uma ou mais letras; `'r'` (ou nada) abre para leitura, `'w'` abre para escrita, `'a'` abre para escrita, com dados escritos acrescentados ao final do arquivo. Se um símbolo `'+'` for agregado ao modo, o arquivo pode ser lido e escrito simultaneamente.

Métodos do objeto arquivo O objeto arquivo possui um conjunto de métodos úteis; os mais importantes são descritos abaixo. Note que o arquivo possui um conceito de posição atual: em um dado momento, operações serão realizadas com base em uma certa posição. Alguns métodos utilizam ou alteram esta posição; outros são operações globais, independentes da posição dela.

- `read()`: como visto acima, retorna uma string única com todo o conteúdo do arquivo.
- `readline()`: retorna a próxima linha do arquivo, e incrementa a posição atual.
- `readlines()`: retorna todo o conteúdo do arquivo em uma lista, uma linha do arquivo por elemento da lista.
- `write(data)`: escreve a string `data` para o arquivo, na posição atual ou ao final do arquivo, dependendo do modo de abertura. Esta função falha se o arquivo foi aberto com modo `'r'`.
- `seek(n)`: muda a posição atual do arquivo para o valor indicado em `n`.
- `close()`: fecha o arquivo.

Qualquer arquivo pode ser aberto e lido desta forma; experimente com esta função, abrindo alguns arquivos locais, lendo e modificando-os.

3.3.2 Leitura do teclado: `raw_input()`

Outra função útil sobretudo para programas Python interativos é a função `raw_input`: lê do teclado uma string, e a retorna. Esta função possui um parâmetro opcional, que é uma mensagem a ser fornecida ao usuário. O exemplo seguinte utiliza um módulo com o nome `leitura.py`:

```
a = raw_input("Entre com um número de 0 a 10: ")
n = int(a)
if not 0 < n < 10:
    print "Número inválido."
if n % 2 == 0:
    print "É Par"
else:
    print "É Ímpar"
```

Este exemplo, quando executado e sendo fornecido o número 7, gera a seguinte saída:

```
>>> import leitura
Entre com um número de 0 a 10: 7
É Ímpar
```

3.4 Orientação a Objetos

Embora termos importantes como classes, objetos e módulos tenham sido introduzidos anteriormente, ainda não discutimos em detalhes os conceitos e a implementação de orientação a objetos (OO) em Python. Python suporta orientação a objetos utilizando um modelo flexível e particularmente homogêneo, que simplifica a compreensão dos mecanismos OO fundamentais da linguagem.

3.4.1 Conceitos de orientação a objetos

Orientação a objetos é um termo que descreve uma série de técnicas para estruturar soluções para problemas computacionais. No nosso caso específico, vamos falar de **programação OO**, que é um paradigma de programação no qual um programa é estruturado em objetos, e que enfatiza os aspectos abstração, encapsulamento, polimorfismo e herança[12].

Convencionalmente, um programa tem um fluxo linear, seguindo por uma função principal (ou o corpo principal do programa, dependendo da linguagem de programação) que invoca funções auxiliares para executar certas tarefas à medida que for necessário. Em Python é perfeitamente possível programar desta forma, convencionalmente chamada de **programação procedural**.

Programas que utilizam conceitos OO, ao invés de definir funções independentes que são utilizadas em conjunto, dividem conceitualmente o 'problema' (ou **domínio**[13]) em partes independentes (**objetos**), que podem conter **atributos** que os descrevem, e que implementam o comportamento do sistema através de funções definidas nestes objetos (**métodos**). Objetos (e seus métodos) fazem referência a outros objetos e métodos; o termo 'envio de mensagens' é utilizado para descrever a comunicação que ocorre entre os métodos dos diferentes objetos.

Na prática, um programa orientado a objetos em Python pode ser descrito como um conjunto de classes — tanto pré-definidas quanto definidas pelo usuário — que possuem atributos e métodos, e que são **instanciadas** em objetos, durante a execução do programa. A seção seguinte concretiza estes conceitos com exemplos.

[12] Mais detalhes sobre conceitos fundamentais de OO podem ser obtidos em `http://en.wikipedia.org/wiki/Object-oriented_programming`

[13] O 'problema' ou 'domínio' de um software compreende o conjunto de tarefas essenciais que este deve realizar; por exemplo, o domínio de um editor de textos compreende escrever, corrigir e alterar documentos — e um conceito fundamental, muito provavelmente modelado em uma classe OO, seria o Documento.

3.4.2 Objetos, classes e instâncias

Objetos são a unidade fundamental de qualquer sistema orientado a objetos. Em Python, como introduzido na seção 3.2.3, *tudo é um objeto* — tipos, valores, classes, funções, métodos e, é claro, instâncias: todos possuem atributos e métodos associados. Nesta seção serão descritas as estruturas da linguagem para suportar objetos *definidos pelo programador*.

Classes Em Python, a estrutura fundamental para definir novos objetos é a **classe**. Uma classe é definida em código-fonte, e possui nome, um conjunto de atributos e métodos.

Por exemplo, em um programa que manipula formas geométricas, seria possível conceber uma classe denominada `Retangulo`:

- Esta classe possuiria dois atributos: `lado_a` e `lado_b`, que representariam as dimensões dos seus lados.
- A classe poderia realizar operações como `calcula_area` e `calcula_perimetro`, e que retornariam os valores apropriados.

Um exemplo de uma implementação possível desta classe está no módulo `retangulo.py` a seguir:

```
class Retangulo:
    lado_a = None
    lado_b = None

    def __init__(self, lado_a, lado_b):
        self.lado_a = lado_a
        self.lado_b = lado_b
        print "Criando nova instância Retangulo"

    def calcula_area(self):
        return self.lado_a * self.lado_b

    def calcula perímetro(self):
        return 2 * self.lado_a + 2 * self.lado_b
```

Esta classe define os dois atributos descritos acima, e três métodos. Os três métodos definidos incluem *sempre*, como primeiro argumento, uma variável denominada **self**, que é manipulada no interior do método. Este é um ponto fundamental da sintaxe Python para métodos: o primeiro argumento é especial, e convenciona-se utilizar o nome self para ele; logo a seguir será discutido para que existe.

Dois dos métodos codificados correspondem às operações descritas inicialmente, e há um método especial incluído: `__init__()`. O nome deste método tem significância particular em Python: é o método **construtor**, um método **opcional** invocado quando a classe é **instanciada**, que é o nome dado à ação de criar objetos a partir de uma classe.

Instâncias A instância é objeto criado com base em uma classe definida. Este conceito é peculiar, e leva algum tempo para se fixar. Uma descrição abstrata da dualidade classe-instância: a classe é apenas uma matriz, que especifica objetos, mas que não pode ser utilizada diretamente; a instância representa o objeto concretizado a partir de uma classe. Eu costumo dizer que a classe é 'morta', existindo apenas no código-fonte, e que a instância é 'viva', porque durante a execução do programa são as instâncias que de fato 'funcionam' através da invocação de métodos e manipulação de atributos.

Conceitualmente, a instância possui duas propriedades fundamentais: a classe a partir da qual foi criada, que define suas propriedades e métodos padrão, e um **estado**, que representa o conjunto

de valores das propriedades e métodos definidos naquela instância específica. A instância possui um **ciclo de vida**: é criada (e neste momento seu construtor é invocado), manipulada conforme necessário, e destruída quando não for mais útil para o programa. O estado da instância evolui ao longo do seu ciclo de vida: seus atributos são definidos e têm seu valor alterado através de seus métodos e de manipulação realizada por outros objetos.

> O que Python chama de 'instância' é freqüentemente denominado 'objeto' em outras linguagens, o que cria alguma confusão uma vez que *qualquer* dado em Python pode ser considerado um 'objeto'. Em Python, instâncias são objetos **criados a partir de uma classe definida pelo programador**.

Retomando o nosso exemplo acima: a partir da classe `Retangulo` que foi definida, poderíamos instanciar objetos retângulo específicos: um com lados de comprimento 1 e 2, e outro com lados de comprimento 2 e 3:

```
>>> from retangulo import Retangulo
>>> r1 = Retangulo(1, 2)
Criando nova instância Retângulo
>>> r2 = Retangulo(2, 3)
Criando nova instância Retângulo
```

Observe que ao instanciar o retângulo:

- Foi importado e utilizado o nome da classe seguido de parênteses.
- Foram fornecidos como argumentos — entre parênteses — dois valores, correspondendo aos comprimentos dos lados diferentes dos retângulos (1 e 2 para o primeiro retângulo, e 2 e 3 para o segundo).
- Estes argumentos são passados — transparentemente — para o método construtor da classe `Retangulo`. O código do método está reproduzido aqui para facilitar a leitura:

```
    def __init__(self, lado_a, lado_b):
        self.lado_a = lado_a
        self.lado_b = lado_b
        print "Criando nova instância Retangulo"
```

Aqui cabe uma pausa para revelar o propósito da variável `self`, definida como primeiro argumento dos métodos. Esta variável representa *a instância sobre a qual aquele método foi invocado.* Esta propriedade é de importância fundamental para OO em Python, porque através desta variável é que atributos e métodos desta instância podem ser manipulados no código dos seus métodos.

Continuando com a análise do bloco de código acima:

- Nas duas primeiras linhas do método — onde é feita atribuição — o código do construtor está atribuindo valores para dois atributos, `lado_a` e `lado_b`, *na instância*, aqui representada pelo argumento `self`. Neste momento, o **estado** da instância passa a conter os dois atributos novos.
- O construtor inclui uma instrução `print` didática que imprime uma mensagem para demonstrar que foi executado; a mensagem aparece na saída do interpretador.

Uma vez instanciados os retângulos, podemos acessar seus métodos. De maneira idêntica aos métodos da lista apresentados na seção 3.2.3, a sintaxe utiliza um ponto seguido do nome do método acompanhado de parênteses:

```
>>> print r1.calcula_area()
2
>>> print r2.calcula_perimetro()
10
```

Conforme esperado, as funções retornaram os valores apropriados para cada instância. Fazendo mais uma demonstração do uso do argumento `self`, vamos observar o código de um dos métodos:

```
def calcula_area(self):
    return self.lado_a * self.lado_b
```

O onipresente argumento `self` aqui é utilizado como meio de acesso aos atributos `lado_a` e `lado_b`. Este código permite visualizar o funcionamento pleno deste mecanismo: ao ser invocado o método `calcula_area` sobre a instância `r1`, o argumento `self` assume como valor esta mesma instância; portanto, acessar atributos de `self` internamente ao método equivale, na prática, a acessar atributos de `r1` externamente.

Em Python é possível, inclusive, acessar os atributos da instância diretamente, sem a necessidade de usar um método:

```
>>> print r1.lado_a
1
>>> print r1.lado_b
2
```

Os valores, logicamente, correspondem aos inicialmente fornecidos à instância por meio do seu construtor.

Atributos Privados e Protegidos Algumas linguagens permitem restringir acesso aos atributos de uma instância, oferecendo o conceito de **variável privada**. Python não possui uma construção sintática literalmente equivalente, mas existem duas formas de indicar que um atributo não deve ser acessado externamente:

- A primeira forma é implementada por meio de uma convenção, não havendo suporte específico na linguagem em si: convenciona-se que atributos e métodos cujo nome é iniciado por um sublinhado (como `_metodo_a`) não devem ser acessados externamente em situações 'normais'.
- A segunda forma estende esta convenção com suporte no próprio interpretador: métodos e atributos cujo nome é iniciado por dois sublinhados (como `__metodo_a`) são considerados de fato privados, e têm seus nomes alterados de maneira transparente pelo interpretador para assegurar esta proteção. Este mecanismo é descrito em maior detalhes na seção *Private Variables* do tutorial Python.

3.4.3 Herança

Um mecanismo fundamental em sistemas orientados a objetos modernos é **herança**: uma maneira de derivar classes novas a partir da definição de classes existentes, denominadas neste contexto **classes-base**. As classes derivadas possuem acesso transparente aos atributos e métodos das classes base, e podem redefinir estes conforme conveniente.

Herança é uma forma simples de promover reuso através de uma generalização: desenvolve-se uma classe-base com funcionalidade genérica, aplicável em diversas situações, e definem-se subclasses concretas, que atendam a situações específicas.

Classes Python suportam herança simples e herança múltipla. Os exemplos até agora evitaram o uso de herança, mas nesta seção é possível apresentar a sintaxe geral para definição de uma classe:

```
class nome-classe(base1, base2, ..., basen):
    atributo-1 = valor-1
    .
    .
    atributo-n = valor-n

    def nome-método-1(self, arg1, arg2, ..., argn):
        # bloco de código do método
    .
    .
    def nome-método-n(self, arg1, arg2, ..., argn):
        # bloco de código do método
```

Como pode ser observado acima, classes base são especificadas entre parênteses após o nome da classe sendo definida. Na sua forma mais simples:

```
class Foo:
    a = 1
    def cheese(self):
        print "cheese"

    def foo(self):
        print "foo"

class Bar(Foo):
    def bar(self):
        print "bar"

    def foo(self):              # método redefinido
        print "foo de bar!"
```

uma instância da classe `Bar` tem acesso aos métodos `cheese()`, `bar()` e `foo()`, este último sendo redefinido localmente:

```
>>> b = Bar()
>>> b.cheese()
cheese
>>> b.foo()         # saída demonstra método redefinido
foo de bar!         # em Bar
>>> b.bar()
foo
>>> print b.a       # acesso transparente ao atributo
1                   # definido em Foo
```

enquanto uma instância da classe `Foo` tem acesso apenas às funções definidas nela, `foo()` e `cheese`:

```
>>> f = Foo()
>>> f.foo()
foo
```

Invocando métodos de classes-base Para acessar os métodos de uma classe-base, usamos uma construção diferente para invocá-los, que permite especificar qual classe armazena o método sendo chamado. Seguindo o exemplo, vamos novamente a redefinir o método `Bar.foo()`:

```
class Bar(Foo):
    # ...
    def foo(self):
        Foo.foo(self)
        print "foo de bar!"
```

Nesta versão, o método `foo()` inclui uma chamada ao método `Foo.foo()`, que conforme indicado pelo seu nome, é uma referência direta ao método da classe base. Ao instanciar um objeto desta classe:

```
>>> b = Bar()
>>> b.foo()
foo
foo de bar!
```

pode-se observar que são executados ambos os métodos especificados. Este padrão, aqui demonstrado de forma muito simples, pode ser utilizado em situações mais elaboradas; seu uso mais freqüente é para invocar, a partir de um construtor de uma classe, o construtor das suas classes-base.

Funções Úteis Há duas funções particularmente úteis para estudar uma hierarquia de classes e instâncias:

- `isinstance(objeto, classe)`: retorna verdadeiro se o objeto for uma instância da classe especificada, ou de alguma de suas subclasses.
- `issubclass(classe_a, classe_b)`: retorna verdadeiro se a classe especificada como `classe_a` for uma subclasse da `classe_b`, ou se for a própria `classe_b`.

Atributos de classe *versus* atributos de instância Uma particularidade em Python, que deriva da forma transparente como variáveis são acessadas, é a distinção entre atributos definidos em uma classe, e atributos definidos em uma instância desta classe. Observe o código a seguir:

```
class Foo:
    a = 1
```

A classe acima define uma variável `a` com o valor 1. Ao instanciar esta classe,

```
>>> f = Foo()
>>> print f.a
1
```

observa-se que a variável parece estar definida na instância. Esta observação convida a algumas indagações:

- Onde está definida a variável `a` – na classe ou na instância?
- Se atribuirmos um novo valor a `f.a`, como abaixo:

  ```
  >>> f.a = 2
  ```

estamos alterando a classe ou a instância?

- Uma vez atribuído o novo valor, que valor aparecerá para o atributo `a` no próximo objeto instanciado a partir de `Foo`?

As respostas para estas perguntas são todas relacionadas a um mecanismo central em Python, que é o **protocolo** `getattr`. Este protocolo dita como atributos são transparentemente localizados em uma hierarquia de classes e suas instâncias, e segue a seguinte receita:

1. Ao acessar um atributo de uma instância (por meio de uma variável qualquer ou `self`) o interpretador tenta localizar o atributo no estado da instância.
2. Caso não seja localizado, busca-se o atributo na classe da instância em questão. Por sinal, este passo é o que permite que métodos de uma classe sejam acessíveis a partir de suas instâncias.
3. Caso não seja localizado, busca-se o atributo entre as classes base definidas para a classe da instância.
4. Ao **atribuir** uma variável em uma instância, este atributo é sempre definido no estado local da instância.

Uma vez compreendido este mecanismo, é possível elucidar respostas para as questões acima. No exemplo, a variável `a` está definida na classe `Foo`, e pelo ponto 2 acima descrito, é acessível como se fosse definida pela própria instância. Ao atribuir um valor novo a `f.a`, estamos definindo uma nova variável `a` no estado local da variável `f`, o que não tem nenhum impacto sobre a variável `a` definida em `Foo`, nem sobre novas instâncias criadas a partir desta.

Se o descrito acima parece confuso, não se preocupe; o mecanismo normalmente funciona exatamente da maneira que se esperaria de uma linguagem orientada a objetos. Existe apenas uma situação 'perigosa', que ocorre quando usamos atributos de classe com valores mutáveis, como listas e dicionários.

```
class Foo:
    a = [1,2]
```

Nesta situação, quando criamos uma instância a partir de Foo, a variável a pode ser **alterada** por meio desta instância. Como não foi realizada **atribuição**, a regra 4 descrita acima não se aplica:

```
>>> f = Foo()
>>> f.a.append(3)
>>> g = Foo()
>>> print g.a
[1, 2, 3]
```

e a variável da classe é de fato modificada. Esta particularidade é freqüentemente fonte de bugs difíceis de localizar, e por este motivo se recomenda fortemente que **não se utilize variáveis de tipos mutáveis em classes**.

3.4.4 Introspecção e reflexão

Introspecção e reflexão são propriedades de sistemas orientados a objetos que qualificam a existência de mecanismos para descobrir e alterar, em tempo de execução, informações estruturais sobre um programa e objetos existentes neste.

Python possui tanto características introspectivas quanto reflexivas. Permite obter em tempo de execução informações a respeito do tipo dos objetos, incluindo informações sobre a hierarquia de

classes. Preserva também **metadados** que descrevem a estrutura do programa sendo executado, e permitindo que se estude como está organizado este sem a necessidade de ler o seu código-fonte.

Algumas funções e atributos são particularmente importantes neste sentido, e são apresentadas nesta seção:

- `dir(obj)`: esta função pré-definida lista todos os nomes de variáveis definidos em um determinado objeto; foi apresentada anteriormente como uma forma de obter as variáveis definidas em um módulo, e aqui pode ser descrita em sua glória completa: descreve o conteúdo de qualquer objeto Python, incluindo classes e instâncias.
- `obj.__class__`: este atributo da instância armazena o seu objeto classe correspondente.
- `obj.__dict__`: este atributo de instâncias e classes oferece acesso ao seu **estado** local.
- `obj.__module__`: este atributo de instâncias e classes armazena uma string com o nome do módulo do qual foi importado.
- `classe.__bases__`: esta atributo da classe armazena em uma tupla as classes das quais herda.
- `classe.__name__`: este atributo da classe armazena uma string com o nome da classe.

3.5 Alguns módulos importantes

Há um grande conjunto de módulos que se instalam juntamente com o interpretador Python; são descritos nesta seção alguns dos mais interessantes.

- `sys`: oferece várias operações referentes ao próprio interpretador. Inclui: `path`, uma lista dos diretórios de busca de módulos do python, `argv`, a lista de parâmetros passados na linha de comando e `exit()`, uma função que termina o programa.
- `time`: oferece funções para manipular valores de tempo. Inclui: `time()`, uma função que retorna o *timestamp*[14] atual; `sleep(n)`, que pausa a execução por n segundos; e `strftime(n)`, que formata um timestamp em uma string de acordo com um formato fornecido.
- `os`: oferece funções que referem-se ao ambiente de execução do sistema. Inclui: `mkdir()`, que cria diretórios; `rename()`, que altera nomes e caminhos de arquivos; e `system`, que executa comandos do sistema.
- `os.path`: oferece funções de manipulação do caminho independente de plataforma. Inclui: `isdir(p)`, que testa se `d` é um diretório; `exists(p)`, que testa se `p` existe; `join(p,m)`, que retorna uma string com os dois caminhos `p` e `m` concatenados.
- `string`: oferece funções de manipulação de string (que também estão disponíveis como métodos da string). Inclui: `split(c, s, p)`, que divide a string `c` em até `p` partições separadas pelo símbolo `s`, retornando-as em uma lista; `lower(c)`, que retorna a string `c` convertida em minúsculas; e `strip(c)`, que retorna `c` removendo espaços e quebras de linha do seu início e fim.
- `math`: funções matemáticas gerais. Inclui funções como `cos(x)`, que retorna o cosseno de `x`; `hypot(x, y)`; que retorna a distância euclidiana entre `x` e `y`; e `exp(x)`; que retorna o exponencial de `x`.
- `random`: geração de números randômicos. Inclui: `random()`, que retorna um número randômico entre 0 e 1; `randrange(m,n)`, que retorna um randômico entre `m` e `n`; `choice(s)`, que retorna um elemento randômico de uma seqüência `s`.

[14] O número de segundos desde 1^o de janeiro, 1970, que por sinal é a data padrão do início do tempo no Unix.

- `getopt`: processamento de argumentos de comando de linha; ou seja, os parâmetros que passamos para o interpretador na linha de execução. Inclui: `getopt()`, que retorna duas listas, uma com argumentos e outra com opções da linha de comando.
- `Tkinter`: um módulo que permite a criação de programas com interface gráfica, incluindo janelas, botões e campos texto.

A documentação do Python inclui uma descrição detalhada (e muito boa) de cada um destes módulos e de seus membros.

3.5.1 Módulos independentes

Além dos módulos distribuídos com o Python, existem vários módulos auxiliares. Justamente por serem numerosos e independentemente fornecidos, não é possível descrevê-los na sua totalidade; vou apenas citá-los; podem ser obtidas maiores informações nos links providos.

- **win32pipe**: permite, na plataforma Windows, executar programas win32 e capturar sua saída em uma string para manipulação posterior. Acompanha a distribuição Activestate Python: `http://www.activestate.com/Products/ActivePython/`.
- **PIL**: Python Imaging Library, que oferece funções para processamento, manipulação e exibição de imagens. `http://www.pythonware.com/products/pil/`
- **NumPy**: provê mecanismos simples e de alto desempenho para manipular matrizes multidimensionais; ideal para operações numéricas de alto volume que necessitem de velocidade. `http://numpy.sourceforge.net/`
- **HTMLgen**: uma biblioteca de classes que gera documentos HTML conforme padrões pré-definidos. Oferece classes para manipular tabelas, listas, e outros elementos de formatação. `http://starship.python.net/crew/friedrich/HTMLgen/html/`
- **DB-API**: Database Application Programming Interface; na realidade, um conjunto de módulos que acessam bases de dados de uma forma padronizada. A API especifica uma forma homogênea de se fazer consultas e operações em bases de dados relacionais (SQL); diversos módulos implementam esta API para bases de dados específicas.
  `http://www.python.org/topics/database/`
- **mx**: oferece uma série de extensões à linguagem, incluindo operações complexas de data e hora, funções nativas estendidas, e ferramentas para processamento de texto.
  `http://www.egenix.com/files/python/`
- **PyGTK**: É outro pacote que permite construir aplicações gráficas com o Python; pode ser usado em conjunto com o Glade, um construtor visual de interfaces.
  `http://www.pygtk.org/`
- **wxPython**: uma biblioteca de classes que permite construir aplicações gráficas multiplataforma usando Python. Há um construtor visual de interfaces disponível, o Boa Constructor. `http://www.wxpython.org/`

Todos os módulos citados se comportam como módulos Python 'normais'; são utilizados por meio da instrução `import`, e boa parte possui documentação e símbolos internos listáveis.

Esta não é uma lista exaustiva, e há muitos outros módulos úteis; há boas referências que listam módulos externos, incluindo o índice de pacotes oficial PyPI: `http://www.python.org/pypi`.

3.6 Fechamento

Aqui termina este tutorial, que cobre os aspectos fundamentais da linguagem Python. Com base no texto, é possível enfrentar uma tarefa prática, que irá exercitar seu conhecimento e ampliar sua experiência. Escolha um problema, e tente passar da sua modelagem para Python.

Chapter 4

Ruby: Another Gem of a Language

Simon Cozens

4.1 Introduction to Ruby

Ruby is one of the new breed of dynamic languages which has been growing in popularity in the West in recent years. In this short tutorial, we're going to look at some introductory principles of Ruby, drawing out some of the interesting features of the language and showing how it can be used to rapidly develop useful tools.

4.1.1 What does Ruby look like?

When you want to get to know someone, you don't start by examining parts of their body, researching their life story or even talking to them. All these things are good, but they come later. You start by just taking a look at them, probably from some distance.

We're going to meet Ruby in this tutorial and so the same things apply. We'll begin by just looking at some Ruby code, from a relatively distant viewpoint.

This is an abridged version of some of the code from the Ruby standard library. It implements complex numbers as a Ruby data type.

```
class Complex < Numeric

  def initialize(a, b = 0)
    @real = a
    @image = b
  end

  def Complex.generic?(other)
    other.kind_of?(Integer) or
    other.kind_of?(Float) or
    (defined?(Rational) and other.kind_of?(Rational))
  end

  def + (other)
    if other.kind_of?(Complex)
      re = @real + other.real
      im = @image + other.image
```

```
    Complex(re, im)
  elsif Complex.generic?(other)
    Complex(@real + other, @image)
  else
    x , y = other.coerce(self)
    x + y
  end
end
```

Already we can draw out a few features from this code which we're going to look at in more depth later:

- Ruby is a freeform language. It does not use braces to delimit scopes, nor does it use semicolons to end statements, although it can. Newline is a perfectly good statement terminator. This gives Ruby code an open, spacious feel.
- Ruby is an object-oriented language. Our new `Complex` class is defined to inherit from the built-in `Numeric` class. Everything in that class is a method.
- Ruby supports defining arguments to methods, as well as providing defaults to those arguments.
- Attributes private to an individual object are defined with the "`@`" sign, much to the surprise of Perl programmers.
- There's a distinction between class methods and object methods. Class methods are defined by fully qualifying them with their class name. Method calls are introduced with a "`.`" preceded either by the object or the class name.
- Class names are first-class objects in Ruby, and can be passed explicitly to methods.
- Operators are just methods. By defining our own methods with the same name as recognised operators, we can override those operators for our objects. Ruby makes heavy use of operator overloading.
- Ruby uses Java-style constructors: "`Complex(1,2)`" returns a new `Complex` object. Additionally, the `new` method is implicit, inherited from the base class `Object` and specialised by `initialize`.

We've talked a lot about objects, because in Ruby everything is either an object or a class. Ruby is a pure OO language, although its syntax is designed to hide this where necessary.

That's the 20,000ft view of Ruby. Now it's time to slow down and look at where Ruby came from and how it's made up.

4.1.2 Where does Ruby come from?

There are two possible ways to answer this question. The first is to say that Ruby came from Japan, and came from the mind of a man called Yukihiro Matsumoto.

Ruby is often described as being a new language, but it's actually around 11 years old now. Part of the reason for this misconception is due to the fact that although it has long had an established user base in Japan, it only became widely known in the West in the past few years.

But that's not really what we mean by this question. We're really asking about Ruby's genesis in the family tree of programming languages.

Ruby draws inspiration from Perl, especially in terms of its flexibility and lack of linguistic dogmatism. However, as Ruby has developed it has diverged from Perl in terms of syntax and semantics. Ruby's object oriented model is drawn from Python and Smalltalk, and its functional features taken from an academic language called CLU.

In this way, Ruby can be thought of as a more OO Perl, a more Perlish Python and a more practical Smalltalk.

4.1.3 Why Ruby?

New scripting languages these days are very common; there are several being produced every year. Why should you learn Ruby?

First, I think it's a particularly good language to begin programming with, since it contains so many elements of different programming styles in a choesive manner and without too much "scaffolding". You can use Ruby in an OO style without wrapping your entire program in superfluous classes and methods; you can use Ruby in a functional style with the benefit of procedural syntax; you can use as much or as little of Ruby's different flavours as you want.

From a Perl point of view, Ruby is particularly interesting because of its influence on Perl 6. If you want to know what Perl 6 is going to look like, start writing Ruby code today. Just as Ruby borrowed heavily from Perl 5 in its design, Perl itself plans to re-assimilate the "good parts" of Ruby back into Perl 6.

More philosophically, every new programming language you learn should change the way you approach programming, or the language is not worth learning. I believe that Ruby has affected the way I think about programming, and have on occasion used Ruby to prototype a complete replacement for existing Perl or Java code.

Part of the reason for this is because programming in Ruby is a lot of fun! It takes Perl's principle of making easy things easy and hard things possible to extremes, and makes things that are easy to *express* into things that are easy to *program*. Ruby fits my brain.

For example, while writing some code, I needed to express the following:

> If there are any elements in array A which also appear in array B, take them out of array A, and put them in array C.

This is easy to say, but in most languages very difficult to code. Here is the best I came up with in Perl:

```
my @intersect = _set_intersection(@a, @b);
if (@intersect) {
    @a = _set_difference(\@a, \@intersect);
    push @c, @intersect;
}

sub _set_intersection {
    my %union; my %isect;
    for (@_) { $union{$_}++ && ($isect{$_}=$_) }
    return values %isect;
}

sub _set_difference {
    my ($a, $b) = @_;
    my %seen; @seen{@$b} = ();
    return grep { !exists $seen{$_} } @$a;
}
```

I actually wrote that after prototyping the following code in Ruby:

```
intersect = a & b
if (intersect.length > 0)
    a -= intersect
```

```
    c += intersect
end
```

This fits the way I think, and I hope it fits the way you think too.

4.2 Operators and control structures

Programmers coming from C, Perl or other ALGOL-derived languages will not be surprised by the majority of Ruby's operators and control structures: the addition operator is "`+`" and this adds two numbers together; there is an "`if-else-end`" statement like so many other languages, and so on. In this section, we want to look at the operators and control structures which make Ruby stand out, not those which make it similar to other languages.

We have already mentioned Ruby's liberal use of operator overloading and its functional credentials, but now it is time to see these in more detail.

4.2.1 Operators

We'll begin with the operators. If you don't see your favourite operator in this section, it's because it does precisely what you'd expect it to do; we're just going to focus on those operators which do something less obvious, but hopefully more useful. Starting with the humble equality comparison operator.

Concepts of Equality

Most languages have one way of comparing whether two things are equal. Perl has two, for string equality and numeric equality. Ruby has three. This can be confusing, until you stop to think what equality means.

Two objects can represent the same data, but be distinct. Two objects can have much in common, but not be equivalent. Two variables can point to exactly the same area of memory. Ruby distinguishes between these three types of equality.

For instance, if we have the two objects:

```
a = Person.new;  b = Person.new;
a.age(19);       b.age(19);
a.name("Sarah"); b.name("Sarah");
```

Are they the same? In one sense, they are. As much as we know about `a` and `b`, they represent exactly the same person. Thus in Ruby, we say:

```
a == b # True!
```

However, they don't refer to the same object. Hence:

```
a.equal?(b) # False!
```

If we were to assign "`a = b`", then they would indeed "`equal?`" each other.

Finally, there's the equality of concept operator, "`a === b`", which is a user-defined match. You can use this to express any kind of equality you like; you might want, for instance, to have "`a === "Sarah"`" return true. We will meet this operator again later.

String Operators

In terms of string operators, as well as the usual ASCII order comparables, ("`<=>`" and friends) Ruby defines a few more unusual operators.

"`%`" is identical to its Python counterpart, and is the "`printf`" operator; it treats the string on its left as a format to be filled, and takes the value or array of values on its right as the arguments to the format:

```
"%s: %2.02f GBP" % ["Coffee", 3]     # "Coffee: 3.00 GBP"
```

"`*`" is used for repetition: "`"\n" * 3`" will produce three newlines.

There are two operators for string catenation, allowing you to choose your preferred style: "`a + b`" will appease those coming from Python, Java and Javascript, while "`a << b`" will make C++ programmers happy.

> **And for you C++ programmers...**
>
> Ruby also overloads "<<" on `IO` objects to mean output, so this will work as expected:
> `STDOUT << "I love " << "C++";`

Array Operators

We've already seem some of the more unusual array operators: the use of "`&`" to return the intersection of two arrays and "`-`" to return the set difference, for instance, but we'll look at a couple more here. Let's look at the ways to add two arrays together. We'lll start with the following two arrays:

```
a = [1, 2, 3]
b = [3, 4, 5]
```

The most obvious way is with the "`+`" operator:

```
irb(main):001:0> a = [1,2,3]; b = [3,4,5]; a + b
=> [1, 2, 3, 3, 4, 5]
```

For adding individual values, we can use "`<<`". This one modifies "`a`":

```
irb(main):002:0> a << 4
=> [1, 2, 3, 4]
```

However, we have to be a little careful with this. Remembering that "`4`" is an object, (a `Fixnum` object, to be precise) we should strictly say that "`<<`" adds an individual object to the end of an array. This clarifies what is going on when we say:

```
irb(main):003:0> a=[1,2,3]; a << b
=> [1, 2, 3, [3, 4, 5]]
```

As you can see, this has added the array "`b`" as a single `Array` object onto the end of "`a`"'s elements, giving us a nested data structure.

Or we could use the "`|`" operator to give us the union of the two arrays:

```
irb(main):004:0> a=[1,2,3]; a | b
=> [1, 2, 3, 4, 5]
```

Only "<<" so far has modified the array "a" methods. We'll look at methods more thoroughly in the next section, but I want to introduce the "push" method here:

```
irb(main):005:0> a.push(b)
=> [1, 2, 3, [3, 4, 5]]
```

Again we've pushed an array as a single object onto the end of a, instead of the contents of the array. To refer to the contents rather than the array itself, we can use the unary "*" flattening operator:

```
irb(main):006:0> a.push(*b)
=> [1, 2, 3, [3, 4, 5], 3, 4, 5]
```

User-defined operator behaviour

This represents, of course, just the operators defined on some of the more interesting built-in classes. As we've seen with our initial Complex example, you can define your own behaviour for operators in your classes. In the first example, we defined what the Complex class's "+" operator should do.

What might be less obvious is that we can define the behaviour of any operator we like in the built-in classes, too, and even redefine ordinary behaviour. If we want to have a Fixnum "-" operator which divides, then we can. Let's not do that, though.

Instead, let's define some behaviour for that user-defined "===" operator we mentioned earlier. We'll make "some_hash === "Ruby"" return true if some_hash has a key called ""Ruby"".

Thankfully, Hash has a "has_key?" method, so this turns out to be quite simple:

```
class Hash
  def ===(key)
      return has_key?(key)
  end
end
```

Now if we have a hash, we can test for the existance of a given key:

```
myhash = { "Ruby" => 10, "Perl" => 5, "Python" => 1 };
myhash === "Ruby" # True
myhash === "C#"   # False
```

4.2.2 Control Structures

Why would we want to do this? The beauty of the user-defined "===" operator is that, like Perl 6's "smart match" operator, "=~", it is used as the default comparator in Ruby's "case" statement.

case/when

Ruby supports a case statement, but its effect is somewhat different to that of other languages. Ruby's "case"/"when" statement looks like this:

```
case value
when String
   puts "Value is a string!"
when "Hello world!"
   puts "Value is a familiar greeting!"
when myhash
```

```
   puts "Value is the name of a programming language"
end
```

This is syntactic sugar for a simple "`if`"/"`elseif`" statement:

```
if String === value
   puts "Value is a string!"
elsif "Hello world!" === value
   puts "Value is a familiar greeting!"
elsif myhash === value
   puts "Value is the name of a programming language"
end
```

The most important thing to remember here is that the thing to be compared goes on the right of the "`===`" sign - it is the argument to the method call, because the thing on the left has the responsibility of deciding how the comparison is to be done. For instance, here, "`String`" is an object of the `Class` class. `Class` defines the "`===`" operator to test whether the right-hand side is a member of the class. If "`value`" is a member of the `String` class, then this tests true. The `String` class itself defines "`===`" differently, and we've just defined our own "`Hash.===`" as being different again.

for/each/yield

Another useful piece of syntactic sugar is the Ruby "`for`" statement. In Ruby, an `Array` object is opaque - the language itself is not allowed to peek inside the object and "get" at the list of elements in the array, so a traditional "`for`" loop over a list would not work. Instead, "`for`" is, you guessed it, a method.

The method in question is "`each`", and it is used a great deal in Ruby; if your class provides an "`each`" method, you can use the "`for`" syntax:

```
for x in myarray
    puts x
end
```

This is purely syntactic sugar for "`myarray.each { |x| puts x }`".

"`each`" is an example of a type of method called an `iterator`. Typically, an iterator takes a block of code, and visits every member of a structure, running that code on each member in turn.

Iterators work because of two features particular to Ruby. The first is that any method can optionally be passed a block of code. This does not appear in the parameter list by default, but is in its own "space".

The second feature is that the kernel method <methodname>yield</methodname> can be used to call this user-supplied block of code with some parameters. For instance, this very trivial method:

```
def method(param)
    yield(param)
end
```

means "call the code you gave me with the parameter you gave me". A slightly more interesting example comes when we add a method to the `Array` class:

```
class Array
   def each_backwards
       self.reverse.each {|x| yield(x) }
```

```
   end
end
```

The use of iterators and "`yield`" is a key part of Ruby, as we'll see later in the idioms section.

Control structure miscellany

Here are a few other notable things about Ruby's control structures which may differ from other languages you know.

- The familiar "`if`" and "`while`" statements have negative counterparts "`unless`" and "`until`".
- Control structures can be used as expressions where this makes sense. For instance, instead of the ternary operator:

  ```
  x = a ? b : c
  ```

  one can legitimately say in Ruby:

  ```
  x = if a then b else c end
  ```

- The loop and conditional structures can be used as statement modifying clauses. That is, instead of saying:

  ```
  if car.empty?
     car.fill
  end
  ```

  one can say

  ```
  car.fill if car.empty?
  ```

- Ruby has a structured exception system; blocks delimited by "`begin`" and "`end`" can trap exceptions in a "`rescue`" clause:

  ```
  begin
     f = open(path)
  rescue
     puts "#{path} does not exist"
  end
  ```

4.3 Ruby idioms and Real code

This last section is a grab-bag of useful or inspiring pieces of Ruby code; particularly neat solutions, or more involved programs showing how Ruby can get real work done.

4.3.1 Built-in methods for everything...

A friend recently asked for a nicer way to do the following in Perl: to remove all the keys of a hash which had an undefined value attached to them. The best piece of Perl he could come up with at the time was:

```
for (keys %attrs) { delete $attrs{$_} unless defined $attrs{$_} }
```

Of course, I was in a Ruby frame of mind at the time, and so came up with a much nicer solution. Shame he couldn't use it...

```
foo.delete_if { |k,v| v.nil? }
```

"`delete_if`" is an iterator over all the keys and values in a hash. If the attached block returns true for a hash element, it gets deleted.

Similarly, Perl programmers may be familiar with the Schwartzian transform for sorting an array efficiently by a function:

```
@sorted = map  { $_->[0] }
          sort { $a->[1] cmp $b->[1] }
          map  { [ $_, function($_) ] }
          @array;
```

While this becomes second nature after a while, it's hardly the most approachable piece of code. Of course, we could translate this directly into Ruby:

```
sorted = array
          .map  { |x|   [ x, function(x) ] }
          .sort { |a,b| a[1].cmp(b[1]) }
          .map  { |x|   x[0] }
```

But there's a somewhat nicer way to do it:

```
sorted = array.sort_by { |x| function(x) }
```

Purists may scorn Ruby's large number of built-in methods, but at the end of the day, they get that job done quickly, and that's what's important.

4.3.2 Iterators are Finalizers

Ruby has no finalizers. There is no easy way to add an action to be done when an object goes out of scope. For instance, if we have a statement handle for a database transaction, it would be nice to do something with the transaction if the handle goes away without being either committed or rolled back. *Something* should happen to it, preferably automatically.

Even without finalizers, though, there turns out to be a very neat way to achieve this. Since methods can take a block and call it back with "`yield`", we can construct an automatically-committing database transaction like so:

```
def transaction
    sth = self.make_statement_handle();
    yield sth
    sth.commit!
end
```

The user code will look like this:

```
database.transaction do |sth|
   sth.update(...)
end
```

At the end of the block, as the "`sth`" is disposed of, the "`transaction`" method takes over again and "`commit`"s the transaction. It's not a real finaliser, but to all intents and purposes it does the same job.

Indeed, this is how many of the Ruby network and threading libraries handle their finalization:

```
require 'net/pop'
pop = Net::POP3::new("pop.simon-cozens.org");
pop.start("simon", password) do
    pop.each { |mail| puts "Incoming mail", mail.header }
end
```

When the "`start`" block is finished, the POP3 library will send the appropriate commands to disconnect the session. This is, incidentally, a neat way of checking one's pop mail while on the road.