XSI Scripting Using Python

Herman Tulleken (herman@luma.co.za)

November 28, 2007

Contents

I. Introduction to Python Programming

1 Basic Concepts

1.1 Introduction

This is the first half of a two-part introduction to programming in Python, with specific focus on the things you will need to know to script for XSI, assuming you have done no programming before.

In this half, we will look at the very basics of programming: working with text, numbers and lists, and executing statements conditionally or repeatedly. We will see how these can be put together into a simple program.

Along the way, you will be introduced to some general programming terms and concepts. Knowing impressive-sounding words won't make you a better programmer, but it sure makes it easier to read and talk about code, something you might do regularly once you start scripting seriously.

1.1.1 Why Python?

Softimage XSI offers four languages that can be used for scripting: Python, Perl, JavaScript and VB Script. These offer the same functionality, but Python has some advantages:

- Python is used as the language of choice in many computer graphics applications (`http://www.zoteca.com/information/wp/pythonEAI.htm`):
  - Industrial Light & Magic, effects studio of the Star Wars films, uses Python extensively in the computer graphics production process.
  - Disney uses Python for its animation production applications.
  - NASA uses Python in several large projects, including a CAD /CAM system and a graphical work-flow modeler used in planning space shuttle missions.

  Python can be used as the scripting language in software such as Maya, Blender, Ogre3D, Panda3D and Houdini.
- Python can be used for stand-alone applications, including games (see `http://www.pygame.org/`).
- Python provides a clean syntax that is easy to read and quick to type. Most Python programmers grow to resent the syntactic clutter of other languages that employ extra keywords, braces, and brackets.
- If you go hard-core, Python has many features that will make your life easier. These features have exotic names such as *operator overloading*, *generators*, *annotations*, and *list comprehensions*. But these are for advanced users only...

When it comes to XSI, there are two disadvantages:

- There are some special issues with Python in XSI covered in the XSI SDK documentation. These relate to some of Python's more advanced features, and the way the interpreter communicates with XSI.
- Python is not well represented in the examples given in the documentation. Most examples are in VB Script and JavaScript.

These are not unsurmountable problems. The solutions to the special issues are given in the SDK documentation, and in practice these issues pose no problems.

The lack of examples is slightly more annoying. However, the XSI commands have the same names in all languages; it is only how you glue them together that differs slightly between the languages. Once you know Python and some basic programming principles, you will be able to interpret these examples and convert them to Python without much thought.

1.1.2 How does Python work?

interpreted:
interpreter:

When you write a program in C++, you need a compiler that translates your C++ code into machine code. Python, and many other scripting languages, are not compiled to machine code. Instead, a python program is *interpreted* on the fly. A special program, called the *interpreter*, reads in a Python program line-by-line, and executes the commands.

You can also type Python commands directly into the interpreter, and get immediate results. It is generally recognised that this improves productivity over the usual program-compile-run cycle used by compiled languages.

source file:

binary file:

A file that contains Python code in text format is called a *source file*. These files have a `.py` extension. When the interpreter reads a source file for the first time, it first converts the source file to a *binary file*, which is easier to interpret. These are the `.pyc` files that appear after you execute a Python program for the first time. Confusingly, the term “compiled” is often used instead of binary; indeed, the ‘c’ in `.pyc` stands for “compiled”.

The binary files can be distributed independently of the source files, making it possible to distribute programs without revealing the source code. However, the user will still need the Python interpreter.

Softimage XSI uses this very interpreter to allow users to manipulate the application through Python code. As you will see later, the details of this is handled in the background and for the most part does not concern the programmer.

1.2 Primitive Types

data types:
primitive types:

Computers work with different *data types*, such as text, strings, images. Every computer language defines a few data types, called *primitive types*, that are the units of data that can be used to construct more complex data types. In Python, the primitive data types are integers, long integers, floating point numbers, imaginary numbers, strings, and booleans. The following section will explain what these types are.

1.2.1 Strings

strings:

Pieces of text are called *strings*. There are several different methods to mark strings; for now we will only work with strings delimited by single-quotes: `'hello'` and `'world'` are two strings.

We can use the `print` statement to print values to the screen:

```
print 'hello world'
```

variable:

A string can be stored in a *variable*. A variable is simply a named place in the computer’s memory. The name helps us to keep track of what is in that memory position—the interpreter handles the details of finding an open place in the memory

large enough to store our value, and freeing the memory when we are done using it. We need not be concerned with this process.

assignment operator:
assign:
initialise:

We use the *assignment operator* = to store a value in a variable. We say we *assign* the value to the variable. When we assign a value to a variable for the first time, we *initialise* the variable. Once a value is assigned to a variable, we can use the variable instead of the original string to perform operations. Here is an example:

```
>>>s = 'hello world'
>>>print s
hello world
>>>
```

In the above example, the variable is `s`. The name `s` is arbitrary; later we will see that it is important to name variables in a way that is easy to remember and read (i.e. interpret its meaning in the program). Large scripts may have hundreds of variables.

Variables must always be initialised before they are used, otherwise the interpreter will complain:

```
>>>print h

Traceback (most recent call last):
  File "<pyshell#68>", line 1, in -toplevel-
    print h
NameError: name 'h' is not defined
```

error message:

The above message is an *error message*. You will encounter them frequently, and deciphering their meaning is an important skill.

Variables can be reassigned:

```
>>>s = 'hello world'
>>>print s
hello world
>>>s = 'bye world'
>>>print s
bye world
>>>
```

Strings can be concatenated with the + operator.

```
>>>s = 'hello '
>>>t = 'world!'
>>>print s + t
hello world
>>>
```

We have "added" the two strings `'hello '` and `'world!'` to form a new string `'hello world!'`. Notice that we have done this by using the variables in which the strings have been stored.

It is often necessary to extract characters in specific positions from a string. This can be done using the indexing operator `[ ]`.

```
>>>s = 'hello world!'
>>>print s[0]
h
>>>print s[1]
e
>>>print s[2]
```

Operator	Operation	Example	Result
+	addition	`10 + 5`	9
-	subtraction	`10 - 5`	5
*	multiplication	`10 * 5`	50
/	division	`10 / 5`	2
%	modulus	`10 % 3`	1

Table 1.1: List of operators

```
l
>>>
```

index: The number in the square brackets is called the *index*. Index 0 always denotes the first character, 1 the second character, and so on.

We can also extract substrings. To extract the substring from index 4 to 8, for example, we use the following:

```
>>>s = 'hello world!'
>>>print s[4:8]
o wo
>>>
```

slice: slice index: The substring `'o wo'` is called a *slice*, and the numbers 4 and 8 are called *slice indices*.

The function `len(str)` returns the length of the string:

```
>>>s = 'hello world!'
>>>print len(s)
12
>>>
```

Notice that the length includes all characters, including spaces and punctuation marks.

In XSI scripting, object names are represented as strings. String manipulation is often useful when working with similarly named objects. As a result, you will recognise how useful naming conventions can be as you become more familiar with scripting.

1.2.2 Numbers

Python can do some simple numeric calculations. Table 1.1 shows some mathematical operations and their associated Python operators.

Like strings, numbers can be stored in variables.

```
>>> a = 4
>>> print a
4
>>>
```

initialised: Once a variable has been assigned with a value (i.e. *initialised*), it can be used in expressions:

```
>>> a = 10
>>> b = 2
>>> a + b
12
>>> c = a + b
```

```
>>> print c
12
>>>
```

Notice that `c` has been initialised to the expression `a + b`.

! → So far we have ignored the fact that there are different number types. There are four types of numbers:

integers: Whole numbers, such as 0, 1, 45, and 234. Integers range from 0 to $2^{31} - 1$.

long integers: Whole numbers that can fall outside the range of integers. They are denoted by an L at the end of the number. For example, `2147483648L`. Every integer also has a long integer representation, for example `1L`.

floating point: Numbers that have a fraction part, such as 11.1, 3.14, 6.185. Every integer also has a floating point presentation, for example `1.0`.

imaginary numbers: These numbers are multiples of the square root of -1, and are denoted by a `j` at the end of the number, for example `1j`.

Most of the time, you don't need to concern yourself with number types. There is one exception, however. Division between integers always gives a whole number; division involving a floating point number and an integer or two floating point numbers gives a floating point number:

```
>>> 12 / 5
2
>>> 12.0 / 5
2.3999999999999999
>>> 12 / 5.0
2.3999999999999999
>>>12.0 / 6
2.0
```

When dividing by integers, the answer is the *highest integer below* the true answer. This type of division is called *floor division*.

floor division:

1.2.3 Booleans

relational operators:

Python provides *relational operators* that can be used to compare values:

```
>>> 3 < 4
True
>>> 3 > 4
False
>>> 3 == 4
False
>>> 3 == 3
True
>>>
```

The result is always a `True` or `False` value, and is called a boolean. Booleans can be stored in variables, just like numbers and strings:

```
>>>s = 3 < 4
>>>print s
True
>>>
```

<	Smaller than
<=	Smaller than or equal to
>	Larger than
>=	Larger than or equal to
==	Equal to
!=	Not equal to

Table 1.2: Relational Operators

The comparison operators can also be used to compare strings (alphabetically). String comparison is case sensitive, and `'A'` comes after `'z'`.

```
>>>'alpha' < 'beta'
True
>>>'a' < 'aa'
True
>>>'Alpha' < 'beta'
False
>>>'Alpha' < 'Beta'
True
>>>
```

1.2.4 Type conversion

Sometimes we need to make a number of a string, or a string of a boolean. Using special functions, we can do this. Here are some examples:

```
>>>s = '123'
>>>t = '456'
>>>print s + t
'123456'
>>>si = int(s)
>>>ti = int(t)
>>>print si + ti
555
>>>print bool(1)
True
>>>print bool(0)
False
>>>print int(str(4) + str(0))
40
>>>print float(3)
3.0
>>>print long(3)
3L
```

1.2.5 Exercises

1. Use Python expressions to calculate the following:
   (a) $34^2 + 35^2$
   (b) $\frac{73-87}{3}$ (Give both the floor division and true division answers.)
2. What are the values of the following Python expressions?
   (a) `'sphere.kine.loc.pos.x'[0:6]`
   (b) `'sphere.kine.loc.pos.x'[7]`
   (c) `'sphere.kine.loc.pos.x'[0:16] + '.' + 'rot.x'`

(d) `'sphere.kine.loc.pos.x'[0:len('sphere')]`

3. What are the values of the following Python expressions?
   (a) `-1 < 1`
   (b) `True < False`
   (c) `'one' < 'two' == 'two' > 'one'`
   (d) `100 % 7 >= 100 % 6`
   (e) `10 < 100 or 100 < 10`
   (f) `10 < 20 and 50 < 40`
   (g) `1 < 2 < 4`
   (h) `1 < 3 < 2`
   (i) `50 and 60`
   (j) `0 and 60`
4. Write an expression that will multiply the digits of a three digit number `n`. (Hint: convert the number to a string).

1.3 Containers

container:

A *container* is a data type that contains a collection of values. A container allows us to refer to a whole bunch of data with a single name, and extract its contents in a uniform manner, as you will see in the following subsections.

1.3.1 Lists

list:
ordered collection:

A *list* is an ordered collection of items, such as strings or numbers. In an *ordered collection*, every item has a fixed position (first, second, etc.), and duplicate items may exist.

To construct a list we use square brackets:

```
>>>fruit = ['apple', 'pear', 'strawberry']
>>>print fruit
['apple', 'pear', 'strawberry']
>>>
```

Note that a list can be stored in a variable, just like strings and numbers.

Using indexing and slice notation, we can extract specific items and sub-lists (slices) from lists:

```
>>>fruit = ['apple', 'pear', 'strawberry']
>>>print fruit[1]
'pear'
>>>print fruit[0:2]
['apple', 'pear']
>>>print fruit[1:3]
['pear', 'strawberry']
>>>
```

Lists can be concatenated with the plus operator:

```
>>>s = [1, 2, 3]
>>>t = [4, 5, 6]
>>>print s + t
[1, 2, 3, 4, 5, 6]
>>>s += t
>>>print s
[1, 2, 3, 4, 5, 6]
```

Lists with duplicate items can be easily created using the multiplication operator:

```
>>>[1] * 5
[1, 1, 1, 1, 1]
```

We can determine whether a list contains an element using the `in` keyword:

```
>>> 0 in [0, 1, 2]
True
>>> 0 in [4, 5, 6]
False
```

mutable:

Elements of lists can be changed, hence lists are *mutable*:

```
>>>s=[1, 2, 3]
>>>print s
[1, 2, 3]
>>>s[1] = 7
>>>print s
[1, 7, 3]
```

1.3.2 Tuples

tuple:

A *tuple* is similar to a list, i.e. an ordered collection of items. Tuples are denoted with round brackets instead of square brackets:

```
>>>fruit = ('apple', 'pear', 'strawberry')
>>>print fruit
('apple', 'pear', 'strawberry')
>>>fruit = ('apple', 'pear', 'strawberry')
>>>print fruit[1]
'pear'
>>>print fruit[0:2]
('apple', 'pear')
>>>print fruit[1:3]
('pear', 'strawberry')
>>>s = (1, 2, 3)
>>>t = (4, 5, 6)
>>>print s + t
(1, 2, 3, 4, 5, 6)
>>>s += t
>>>print s
(1, 2, 3, 4, 5, 6)
>>>(1) * 5
(1, 1, 1, 1, 1)
```

immutable:

The important difference between lists and tuples is that a tuple cannot change once it has been created, that is, they are *immutable*. The following gives an error:

```
>>>s = (1, 2, 3)
>>>s[1] = 7

Traceback (most recent call last):
  File "<pyshell#73>", line 1, in -toplevel-
    s[2] = 1
TypeError: object does not support item assignment
```

Tuples are frequently employed in XSI. For example, all the children of an object can be obtained as a list, as we will see later.

1.3.3 Exercises

1. What are the values of the following Python expressions?
   (a) `[1, 2, 3, 4, 5, 6][2:5] * 2`
   (b) `['sphere', 'cube', 'prism'][1] + 'cylinder'`
   (c) `[[1, 2, 3], [4, 5, 6], [7, 8, 9]][1][2]`
   (d) `[[1, 2, 3], [4, 5, 6], [7, 8, 9]][1:2]`
   (e) `[[1, 2, 3], [4, 5, 6], [7, 8, 9]][1]`
   (f) `[[1, 2, 3], [4, 5, 6], [7, 8, 9]][1][2]`
   (g) `[[1, 2, 3], [4, 5, 6], [7, 8, 9]][1:2]`
   (h) `[[1, 2, 3], [4, 5, 6], [7, 8, 9]][1]`
   (i) `[[1, 2, 3], [4, 5, 6], [7, 8, 9]][1][2] * 4`
   (j) `[[1, 2, 3], [4, 5, 6], [7, 8, 9]][1:2] * 4`
2. What are the values of the following Python expressions?
   (a) `(0,)`
   (b) `((1, 2) * 3, (3, 4) *2) + (0, 1)`
3. What do the following statements do?
   (a) `a, b = 'hello', 'world!'` (Hint: print the values of `a` and `b` after you executed the preceding statement.)
   (b) `(a, b), c = (1, 2), (3, 4)` Can you explain the result?

1.4 Programs

program:

After this section, we will frequently need to execute several statements. Since it is inconvenient to type several statements at the command line, we will enter them in a file instead, and execute them all at once. Such a collection of statements are called a *program*. Python programs are usually stored in files with extension `py`, and can be loaded from the *File* menu.

comments:

Even for small programs, it is hard to determine from code alone what the program does. To aid human readers, *comments* are added to programs. Comments are ignored by the interpreter, and therefor have no set structure. Comments in Python programs are preceded with a hash `#`. The comment extends to the end of the line. Here is an example:

```
# Hello World program.
# This program prints the string 'Hello World!' to screen.
# author: me (me@mydomain.com)

print 'Hello World!' #this statement prints 'Hello World!'
```

In this simple example comments are redundant, since it is clear what the program does. This is rarely the case in general, and you should comment all your programs.

It is also a good idea to include your name and e-mail. If there is a bug in your code, your team mates will know who might be able to fix it; a zealous user might even fix the bug and send you the update. And if your code is made available to the public, you might be credited if your code is used in another project.

1.4.1 Exercises

1.5 Conditional statements

In many instances we need to execute statements depending on some condition.

1.5.1 The if-statement

if-statement:

The *if-statement* allows us to do this:

```
a = 3
if a < 4:
   print 'smaller'
else:
   print 'bigger'
```

This program will print `'smaller'`. If we changed it to the following program:

```
a = 5
if a < 4:
   print 'smaller'
else:
   print 'bigger'
```

it will print 'bigger' instead. In these examples the expression `a < 4` is the condition. The else statement is not always needed:

```
a = 3
if a > 0:
   print 'positive!'
```

An `if`-statement can have multiple conditions, each tried in succession until one succeeds:

```
a = 3
if a > 0:
   print 'positive!'
elif a < 0:
   print 'negative!'
elif a == 0:
   print 'zero !'
else:
   print 'impossible?'
```

This program prints out `'positive!'`. Change the value of `a` first to `-3`, and then to `0`, and observe the output each time. The keyword `elif` is a contraction of `else if`. The following program is equivalent to the one above, but harder to read:

```
a = 3
if a > 0:
   print 'positive!'
else:
   if a < 0:
      print 'negative!'
   else:
      if a == 0:
         print 'zero !'
      else:
         print 'impossible?'
```

indentation:

Indentation refers to white space before every line of code. Python uses your indentation to interpret your code's structure. Consider the following two examples:

```
s = 3
if s < 2:
   print 'hello'
   print 'world'
```

```
s = 3
if s < 2:
   print 'hello'
print 'world'
```

The two programs produce different output. The first prints nothing; the second prints `'world'`. The reason is that in the first program, the indentation indicates that the second print statement is part of the block that gets executed only if the `if`'s condition is met. In the second program, the second print statement is not indented, so it is always executed.

Giving meaning to indentation is a big departure from other main stream languages, which usually use braces or special keywords to delimit blocks. Here is the first program in C:

```
s = 3;
if(s < 2)
{
   printf("hello");
   printf("world");
}
```

Where to place braces is often fiercely debated on the web; Python programmers are spared this burden.

1.5.2 Exercises

1. Write an `if`-statement that will print whether a number is even or odd (Hint: `x % 2` returns `0` if `x` is even.)
2. Write an `if`-statement that will print whether a string starts with a capital letter or not. (Hint: remember that `'Z' < 'a'` is `True`.)
3. Write an `if`-statement that will print out whether a string is the name of a day, a month, or neither. (Hint: use the `in` keyword with tuples or lists of the days and months.)

1.6 Iteration

iteration:

Iteration is a fancy name for doing something repeatedly, perhaps with some variation. To print all the digits from 0 to 9, we could use the following program:

```
print 0
print 1
print 2
print 3
print 4
print 5
print 6
print 7
print 8
print 9
```

This is repetitive, clumsy, and thus error prone. By using iteration instead, we can construct a very succinct program:

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

loop:
`for`-statement:

The two statements above forms a *loop*. Since the loop is constructed using a *for-statement*, we call the loop a `for`-loop.

To understand how the loop works, first type in the following into the interpreter:

```
>>>range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

The `range` function returns a list containing all the integers from 0 below the given integer.

body:

The `for`-statement takes a list, and assigns each item to the given variable (called the counter), and executes the statement(s) in the *body* of the loop.

Look at the following example:

```
for i in ['apple', 'pear', 'strawberry']:
   print i
   print i[0]
```

The program prints each item followed by its first character:

```
apple
a
pear
p
strawberry
s
```

Here is what is going on:

1. the `for`-statement assigns `'apple'` to `i`, prints `i` (i.e. it prints `'apple'`), and then print the first letter of `i` (i.e. `'a'`);
2. then the `for`-statement assigns `'pear'` to `i`, prints `i` (`'pear'`) and the first letter of `i` (`'p'`);
3. finally, the `for`-statement assigns `'strawberry'` to `i`, prints `i` (`'strawberry'`) and the first letter of `i` (`'s'`).

Notice that the indentation determines which statements form part of the loop's body. Compare the example above with the following:

```
for i in ['apple', 'pear', 'strawberry']:
   print i
print i[0]
```

The output is:

```
apple
pear
strawberry
s
```

The last print statement is executed only once, since it is not part of the `for`-statement's body.

nested:

Loops can be *nested*, that is, one loop can be placed in the body of another. The following program produces a list of multiplication statements.

```
for i in range(10):
   for j in range(10):
      print str(i) + ' * ' + str(j) + ' = ' + str(i*j)
```

1.6.1 The while-statement

`while`-statement:

The *while-statement* is another kind of loop. The while loop is executed as long as its condition is met:

```
i = 0

while i < 10:
   print i
   i += 1
```

Here the body of the while loop is executed as long as the value if `i` is smaller than 10. Since we increment `i` with every iteration, its value must reach 10, at which point the loop stops. While loops are usually employed when we cannot predict how many iterations will be performed; this is often the case when we read files, get data from a network, or perform a search of some kind. The following program checks whether a string contains a vowel:

```
s = 'hello world'
i = 0
vowels = ['a', 'e', 'i', 'o', 'u']
hasVowel = False

while not hasVowel and i < len(s):
   hasVowel = s[i] in vowels
   i += 1

if hasVowel:
   print 'The string has a vowel'
else:
   print 'The string has no vowel'
```

1.6.2 Exercises

1. Write a program that will print all the characters of your name successively. Use a `for`-loop; your program should not be more than two or three lines.
2. Write a program using two `for`-loops that outputs the following:

   ```
   *
   **
   ***
   ****
   *****
   ******
   *******
   ```

   (Hint: experiment with the range function: try `range(1, 8)`.)
3. Can you modify the program above to use one loop instead? (Hint: type `'*' * 4` in your interpreter).
4. Write a program using a loop that outputs the following:

   ```
   *
   &&
   ^^^
   %%%%
   $$$$$
   ######
   @@@@@@@
   ```

(Hint: use the string `' *&^%$#@'`.)

5. Write a program that prints the first letter of a string that comes after 'm' in the alphabet.

1.7 Resources

`http://swaroopch.info/text/Byte_of_Python:Main_Page` A beginner's tutorial.

`http://docs.python.org/tut/` The official Python tutorial.

`http://www.mindview.net/Books/TIPython/` A more advanced book on Python (HTML downloadable).

2 Bigger structures

2.1 Introduction

In this second part of *Introduction to Python*, we look at the programming structures that help to modularise code, and make it possible to write programs with thousands of lines of code without going insane.

You will also be introduced to some of the predefined pieces of code that comes with Python: modules, functions, and methods of built-in types.

Should you know them all?

You should try. The more you know about Python, the more productive you will be. You will work faster, and your work will be useful in more contexts. Hunting down a function that does something specific can take an entire morning; writing your program without that function can take an extra day; rewriting a program for another but similar task can take an extra week. Time is money, and knowledge is time: *you* do the math.

2.2 Functions

You have already met a few functions:

- The `len` function takes a string as its imnput, and gives the length (in characters) of that string as output.
- The `range` function takes an integer, and gives a list of integers ranging from 0 up to but excluding the argument.

function:
argument:
return:
call:

A *function* is a piece of code with a name. In general, a function takes a number of inputs called *arguments*, and *returns* a result that somehow depends on the arguments as its output. Once a function is defined, it can be used over and over. When we use a function, we say we *call* the function. Since functions are called by names, they also serve to document: the function name is an indication of what that function does, and helps the reader of code to figure out what the code does or how it works.

In the next section we will see how to define our own functions.

2.2.1 Defining Functions

The following example defines a function that simply prints its argument:

```
def myPrint(str):
   print str
```

If you execute you program, nothing seems to happen. In the background, however, your function's definitions has been put in memory, so that you can call the function in the interpreter:

```
>>>myPrint('Hello')
Hello
>>>
```

return-statement:
return:

The *return-statement* is used to show what the output of your functions should be. We say a function *returns* a result.

```
def double(x):
   return x * 2
```

In this example, the function double returns its argument times 2. Load your program, and see what happens in the interpreter:

```
>>>double(3)
6
>>>double('hello')
'hellohello'
>>>double (['firstItem', 'secondItem'])
['firstItem', 'secondItem', 'firstItem', 'secondItem']
>>>
```

The `return`-statement forces the interpreter to leave the function. Modify your double function as follows:

```
def double(x):
   return x * 2
   print 'hello'

>>>double(3)
6
>>>
```

As you can see, the last line of the function is not executed, since the `return`-statement forces the interpreter to exit from the function. Sometimes we want to return without a result. In this case, the `return`-statement can be used alone:

```
def safeDivide(x, y):
   if y == 0:
      print 'Can\'t divide by zero!'
      return
   return x / y
```

When defining functions with more than one argument, separate the arguments with commas. The following function adds two numbers together:

```
def myAdd(x, y):
   return x + y
```

2.2.2 Functions as Data

first-class objects:

In Python, functions are *first-class objects*. This means functions are on equal footing with other data types: they can be stored in variables, used as arguments to other functions, and returned as results of functions.

In the following example, a function is assigned to a variable. Notice how we use the variable name to call the function:

```
>>>def double(x): return x*2

>>>double(3)
6
>>>fn = double
>>>fn(3)
6
```

In this example, our function takes a function as one of its arguments, and applies it to the other two arguments before adding them together and returning the result:

```
>>>def coolAdd(fn, a, b): return fn(a) + fn(b)

>>>coolAdd(abs, 2, -2)
4
>>>coolAdd(max, [2, 3, 4], [8, 5, 3])
12
>>>coolAdd(len, 'hello', 'joe')
8
```

The final example illustrates how a function can return a custom-built function:

```
def makeMult(n):
   def fn(x):
      return x * n

   return fn

>>>tripple = makeMult(3)
>>>tripple(5)
15
```

Note that we defined a function *inside* another function.

procedural programming:

functional programming:

Procedural programming is a style of programming where the programmer defines and calls functions. Using functions also as data gives rise to a style of programming called *functional programming*. Although functional programs can be very elegant, they are harder to design; procedural programming is more direct and by far more prevalent.

2.2.3 Built-in functions

Here we discuss some built-in functions that might come in useful. For a complete list, refer to the Python documentation.

`abs(x)`
: Returns the absolute value of `x`, that is, if `x` is negative, `-x` is returned, otherwise `x` is returned. For example

```
>>>abs(-3)
3
>>>abs(4.2)
4.2
>>>
```

`dir(x)`
: Returns a list of the object's or module's attributes is returned. If `x` is omitted, this function returns a list of defined functions and variables.

```
>>> s = 'hello'
>>> dir()
['__builtins__', '__doc__', '__name__', 's']
>>> dir(s)
['__add__', '__class__', '__contains__', '__delattr__',
'__doc__', '__eq__', '__ge__', '__getattribute__',
'__getitem__', '__getnewargs__', '__getslice__', '__gt__',
  '__hash__', '__init__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
```

```
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
'__setattr__', '__str__', 'capitalize', 'center', 'count',
'decode', 'encode', 'endswith', 'expandtabs', 'find',
'index', 'isalnum', 'isalpha', 'isdigit', 'islower',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'replace', 'rfind', 'rindex', 'rjust', 'rsplit',
'rstrip', 'split', 'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper', 'zfill']
```

This function is useful for analysing ill-documented classes and modules, and is also a useful debugging tool.

`divmod(x, y)`

Returns `(x / y, x % y)`:

```
>>>divmod(13, 5)
(2, 3)
```

`eval(x)`

This interprets the string `x` as a Python expression, evaluates it, and returns the result:

```
>>>eval('[3] * 8')
[3, 3, 3, 3, 3, 3, 3, 3]
```

`filter(fn, list)`

Returns the elements of `list` for which `fn` returns `True`.

```
>>>def isint (x): return type(x) is int

>>>filter(isint, [12, 12.0, 'hello', '2', -3]
[12, -3]
```

`map(fn, list, ...)`

This function applies the given function `fn` to each item in the list, and returns the results as a list.

```
>>>map(abs, [-2, -1, 0, 1, 2])
[2, 1, 0, 1, 2]
```

If `fn` takes more than one argument, one list must be given for each argument. For example:

```
>>>map(__add__, [1, 2, 3], [4, 5, 6])
[5, 7, 9]
```

`len(x)`

Returns the length of sequence `x`.

```
>>> len('hallo')
5
>>>len([0, 1, 2])
3
```

`max(x, ...)`

If only one argument is specified, returns the largest element of the sequence, otherwise returns the largest argument.

```
>>>max([1, 3, 2])
3
>>>max(1, 3, 2)
3
```

`min(x, ...)`
: If only one argument is specified, returns the smallest element of the sequence, otherwise returns the smallest argument.

```
>>>min([1, 3, 2])
3
>>>min(1, 3, 2)
3
```

`range(x, y, z)`
: If only `x` is specified, returns the list `[0, 1, 2, ..., x -1]`. If `x` and `y` is specified, the list starts at `x` and ends just before `y`. If all three arguments are specified, the list starts with `x`, ends just before `y`, and elements differ by `z`.

```
>>>range(7)
[0, 1, 2, 3, 4, 5, 6]
>>>range(1, 7)
[1, 2, 3, 4, 5, 6]
>>>range(1, 7, 2)
[1, 3, 5]
```

`reload(module)`
: Reloads a module. This is especially useful in XSI when developing a module. Calling this function at the beginning of the module will reload this module, so that all your changes will take effect.

`sort(seq)`
: Returns a sorted copy of the given iterable object.

```
>>>sorted([5, 3, 1, 2, 4])
[1, 2, 3, 4, 5]
```

`sum(seq, start)`
: If only `seq` is given, this function returns the sum of the elements of the list. If `start` is given as well, the value of `start` is also added to the result:

```
>>>sum([1, 2, 3])
6
>>>sum([1, 2, 3], 10)
16
```

2.2.4 Exercises

1. Implement a function that takes a list and a number as arguments, and returns a list with all elements of the original list greater than the given number.
2. Implement a function that takes a list of strings and an integer `n`, and truncates all strings so that their lengths equal `n`. Strings whose lengths' are smaller than `n` should not be changed.
3. Implement a function that takes a list of strings, an integer `n` and a character `c`, and pads all strings with `c` so that their lengths equal `n`. Strings whose lengths' are larger than `n` should not be changed.
4. Implement a function that takes a list of strings, an integer `n` and a character `c`, and returns a list of strings padded or truncated to length `n`. Use the functions you defined in the previous two exercises.
5. Implement a function `mul` that will multiply all the elements of a list together. Assume that the list contain only numeric elements.
6. Implement a function that will square all elements of a list. Assume that the list contain only numeric elements.

7. Implement a function that takes a single argument `n` and prints a triangle of stars with `n` rows, for example

   ```
   *
   **
   ***
   ```

8. Define a function that takes one argument `x`, and returns a function that takes a list as argument, and filters out all elements of the list equal to `x`.

2.3 Classes

2.3.1 Attributes

class:

A *class* is special data type that can be used to bunch data and functions together. A class has data attributes and method attributes.

data attribute:

Data attributes are the data the class is made of; data attributes are variables that belong to a class. For example, a class `Point` can be made out of three data attributes, `x`, `y` and `z`, each a number representing a coordinate. If `point` is of type `Point`, we can access its attributes with the following notation:

```
>>>print point.x
0
>>>point.x += 10
>>>print point.x
10
```

instance:
instantiation:
constructor:

Values that are of a class type are called *instances* of that class. Creating an instance of a class is called *instantiation*. A class is instantiated by calling a special function: the class's *constructor*.

```
>>>point = Point(0, 0, 0)
```

The class constructor has the same name as the class. What arguments the constructor takes and what it actually does is defined by whomever implemented the class.

attribute methods:

A class can also have special functions associated with it, similar to the way attributes are special variables associated with the class. These functions are called *attribute methods*, and are called using a special syntax:

```
>>> point = Point(10, 20, 30)
>>> point.move(1, 2, 3)
>>> print point.x
11
```

The attributes, constructor, and methods of a class can be looked up in the documentation that comes with the class. For instance, standard library classes are described in the Python module docs, and the classes of XSI SDK is defined in the SDK documentation.

A class instance can be used just like any other variable: they can be assigned, printed, put into lists, and used as arguments and return values of functions. Here is an example:

```
def add(point1, point2):
   Point result = Point(0, 0, 0)
   result.x = point1.x + point2.x
   result.y = point1.y + point2.y
   result.z = point1.z + point2.z
```

```
   return result
```

or equivalently:

```
def add(point1, point2):
   return Point(point1.x + point2.x, \
      point1.y + point2.y, point1.z + point2.z)
```

2.3.2 Inheritance and Polymorphism

You have already seen the effects of inheritance:

```
>>> len('hello')
5
>>> len([0, 0, 0, 0, 0])
5
>>> len((0, 0, 0, 0, 0))
5
```

derived:
sequence:
enherits:

The `len` function works for all these types, because they are all *derived* from a single type: all three types are *sequence* types. We say that the type string *enherits* from type sequence: a string is a *type of* sequence. The same goes for type tuple and list—they too inherit from type sequence. We call type sequence the *base type* of the derived types.

base type:

When one type is derived from another type, the derived type share all the attributes of the base type. For example, type sequence has the `__len__()` method attribute, which can be called from all the derived types:

```
>>>'hello'.__len__()
5
>>>[0, 0, 0, 0, 0].__len__()
5
>>>(0, 0, 0, 0, 0).__len__()
5
```

This is the secret to the `len` function, which could be defined as follows:

```
def len(x):
   return x.__len__()
```

override:

In some cases, the derived type can *override* a method attribute of the base type, that is, it can defined its own specialised version of it. The method `__repr__` is defined for all types. This is used by the print statement to print a representation of a value. The three sequence types `string`, `tuple`, and `list`, all define their own special versions, so that they are printed very differently. The `print` statement calls this method, and and the appropriate `__repr__` method is automatically called. This process, where we make use of the fact that for a variable the appropriate method is called depending on its type, is called *polimorphism*.

polimorphism:

Before the invention of object orientated languages, programmers had to write code somewhat like the following:

```
def myPrint(x):
if type(x) is TypeString:
   printString(x)
elif type(x) is TypeTuple:
   printTuple(x)
elif type(x) is TypeList:
   printList(x)
```

```
...
```

Now, instead, we can write something like this:

```
def myPrint(x):
   printString(x.__repr__())
```

The method `__repr__` automatically calls the right method depending on the variables type, reducing our amount of code considerably.

2.3.3 Mutable Sequence Type Methods

Here is a list of the most useful mutable sequence type methods:

`s.append(x)`
: Appends the sequence `x` to the sequence `s`.

```
>>>s = [1, 2, 3]
>>>s.append([4, 5, 6])
>>>print s
[1, 2, 3, 4, 5, 6]
>>>
```

`s.extend(x)`
: Adds the element x to the end of the list.

```
>>>s = [1, 2, 3]
>>>s.extend(4)
>>>print s
[1, 2, 3, 4]
>>>
```

`s.count(x)`
: Counts the number of elements of `s` that equals `x`.

```
>>>s = [1, 4, 2, 4, 2, 1, 4, 1]
>>>s.count(1)
3
```

`s.index(x, j, k)`
: Returns the smallest index of the list such that the element at that position equals `x`, and the index lies between `j` (inclusive) and `k` (exclusive). If `j` and `k` are omitted, the index returned can be in the range `0 to len(s)`.

```
>>>s = [1, 4, 2, 4, 2, 1, 4, 1]
>>>s.index(4)
1
>>>s.index(4, 2, 5)
3
>>>
```

`s.remove(x)`
: Removes the first occurence of `x` from the list.

`s.reverse()`
: Reverses the list.

`s.sort(cmp, key, reverse)`
: Sorts a list: `cmp` is a two-argument function that is used to compare the elements of the list; `fn` is a one-argument function applied to elements before the comparison is performed; and `reverse` is a boolean indicating whether this list should be reversed or not.

  Given the following definitions:

```
def compStringLength(s1, s2):
   return len(s1) < len(s1)

def getFirst(x):
   return x[0]
```

here are some sort examples:

```
>>>s = [4, 2, 5]
>>>s.sort()
>>>print s
[2, 4, 5]
>>>s = ['joe', 'henry', 'jane', 'sandra']
>>>s.sort(compStringLength)
>>>print s
['joe', 'jane', 'henry', 'sandra']
>>>s = [('joe', 12), ('henry', 4), ('jane', 8), ('sandra', 3)]
>>>s.sort(compStringLength, getFirst)
>>>print s
[('joe', 12), ('jane', 8), ('henry', 4), ('sandra', 3)]
>>>s.sort(compStringLength, getFirst, True)
>>>print s
('sandra', 3), ('henry', 4), ('jane', 8), ('joe', 12)]
```

Here you can get a glimpse of the power of functions as data—the implementers have implement only one sort, but it can be used with any comparison function and by any component of a compound type.

2.3.4 String methods

The string class provides a multitude of useful string operations.

`s.capitalize()`
: Returns a copy of th string with the first character capitalized.

`s.find(sub, j, k)`
: Returns the smallest index between `j` and `k` of the string where the substring `sub` can be found. If `j` and `k` are not provided, the index can range from 0 to the length of the string.

`s.isalnum()`
: Returns True if all the characters in this string are alpha-numeric characters.

`s.isalpha()`
: Returns True if all the characters in this string are alphabetic characters.

`s.isdigit()`
: Returns True if all the characters in this string are digits.

`s.lower()`
: Returns a copy of this string in lower case characters.

`s.split(sep, maxsplit)`
: Returns a list of all strings separated with the character `sep`, up to `maxsplit` strings. If `maxsplit` is not provided, all the strings are provided.

```
>>>'sphere.kine.loc.x'.split('.')
['sphere', 'kine', 'loc', 'x']
>>>'sphere.kine.loc.x'.split('.', 2)
['sphere', 'kine']
>>>
```

`s.strip(chars)`
: Returns a copy of this string stripped of leading and trailing characters in the string `chars`. If `chars` is not provided, the string is stripped of white space:

```
>>>’  hello ’.strip()
’hello’
>>>’Yes...?’.strip(’!?,.’)
’Yes’
```

`s.upper()`
: Returns a copy of this string in uppercase characters.

2.3.5 Exercises

1. Implement a case-insensitive sorting function for a list of strings.
2. Implement a function that removes all punctuation characters from a string.
3. Implement a function that removes all occurrences of a given item from a list.
4. For this exercise you will need the `Object3D` class. This class has four data attributes: `name` is a string name of this object, and `x`, `y` and `z` are the three coordinates of the object's position.

   Implement a function that sorts a list of `Object3D` instances depending to their distance from the origin. (Hint: The distance of an object at position (x, y, z) from the origin is given by $\sqrt{x^2 + y^2 + z^2}$.)

2.4 Scope

scope:

Variable names are only valid in the block in which they have been defined. If this was not the case, it would be necessary to think up and remember a huge number of variable and function names in large programs. The region of code where a variable is valid, is called its *scope*. When execution exits this block, we say the variable is out of scope. In the following example, the variable `x` only has scope in the function:

```
def assignX():
   x = ’hello’

assignX()
print x
```

global scope:

global variable:

If you execute the program, you will get a name error: *name 'x' is not defined*. The outermost scope of a program is called the *global scope*. Variables with global scope can be accessed from any part of you program, and are called *global variables*. You can refer to the global scope with the `global` keyword:

```
def assignX():
   global x
   x = ’hello’

assignX()
print x
```

This time, the message is printed.

Look at the following example:

```
x = ’bye’

def printX():
```

```
   print x

printX()
```

Notice that we do not get an error this time; the program prints `'bye'`. This is because we have not assigned the variable `x`, we have not created a new locally scoped named `x`.

```
x = 'bye'

def printX():
   x = 'hello'
   print x

printX()
print x
```

The program prints `'hello'`, then `bye`. Since we have created a local variable named `x`, we did not modify the global variable `x`. In our function, the global variable is *hidden* by the local variable. The global keyword can prevent this:

hidden:

```
x = 'bye'

def printX():
   global x
   x = 'hello'
   print x

printX()
print x
```

This time, the program prints `'hello'` twice.

2.5 Modules

module:

Modules are, like functions, an organisational structure of code. A *module* is simply what we call the thing that contains all the functions, variables and types defined in a single file[1]. The module name is the file name, without the extension.

2.5.1 Using Modules

You can access module functions, variables and types with the following syntax:

```
from math import sin #import the function sin from the module math
```

import statement:

A statement like this is called an *import statement*, and should occur near the top of your file, before any other code. You can also import modules from the interpreter:

```
>>> sin (1)

Traceback (most recent call last):
  File "<pyshell#12>", line 1, in -toplevel-
    sin(1)
NameError: name 'sin' is not defined
>>>from math import sin
>>>sin(1)
```

[1] Although it is possible, with some trickery, do circumvent this “limitation”. This is more often than not, an abuse of the module concept, and thus not covered here

```
0.8414709848078965
>>>
```

You will sometimes see the following:

```
from math import *
```

This imports *all* the elements of module math. This is useful for working in the interpreter, but should not be used in program files:

- You might unknowingly introduce name clashes. For instance, the function `add` is defined in two standard library modules. Importing everything from both modules may lead you to use add from module `audioloop` while intending to use the one in module `operator`. This will lead to error messages that while baffle you; this kind of bug can be very hard to track down.
- Readers of your code will have to do more to decipher dependencies, or figure out where a particular function is defined. You might import a dozen modules, some of them your own. The poor soul reading your code has to go through all of them to see where that function is defined. Very often this poor soul is yourself, trying a year later to fix a recently discovered bug.

There is a third way to access module elements:

```
import math

print math.sin(1)
print math.cos(1)
```

Although this approach seems to provide a reader with the most information, right where it is used, it actually distracts and should be avoided. This notation is, however, useful for disambiguation:

```
import audioloop
import operator

...

print operator.add(1, 2)
print audioloop.add(fragment1, fragment2, 128)
```

2.5.2 Standard Modules

The Python standard library has some 250 modules defined. For a complete list and comprehensive explanations of each module's functions, variables and types, refer to the Python documentation. Here I will only list the few that are used frequently.

`math`
: This module contains trigonometric functions (`sin`, `cos`), exponential and logarithmic functions (`exp`, `log`), some rounding functions (`floor`, `ceil`) and versions of functions more suitable fo floating point numbers (`fabs`, `fmod`). The module also defines the constants `pi` (3.141592...) and `e` (2.718281...).

`random`
: This module contains functions for generating random numbers (`random`, `randint`), selecting random elements from sequences (`choices`, `sample`), and shuffling sequences (`shuffle`).

`pickle`
: This module is used for *serialising* Python objects. An object is *serialised* if it is written to a file in a way that it can be reconstructed by reading the file

serialise:

again. This is a very convenient way of saving data to file, since you do not have to worry about the details of how the object is represented in a file, and how you should read and write it.

`copy`

This module contains deep and shallow copy operations, and applies to compound objects, like lists and class instances. A *shallow* copy only copies *references* to the contained objects. That is, if you change the value of the contained object, this change will reflect in the shallow copy:

shallow: references:

```
>>>from copy import copy
>>>x = [1, 2, 3]
>>>y = ['a', 'b', 'c']
>>>z = [x, y]
>>>print z
[[1, 2, 3], ['a', 'b', 'c']]
>>>newZ = copy(z) #shallow copy
>>>x[1] = 10
>>>print z
[[1, 10, 3], ['a', 'b', 'c']]
>>>print newZ
[[1, 10, 3], ['a', 'b', 'c']]
```

A *deep copy* copies the values of the contained objects. This is done recursively, that is, if the contained object is also a compound object, its contents is also deep copied. Changes in the contained objects will not reflect in the deep copy:

deep copy:

```
>>>from copy import copy
>>>x = [1, 2, 3]
>>>y = ['a', 'b', 'c']
>>>z = [x, y]
>>>print z
[[1, 2, 3], ['a', 'b', 'c']]
>>>newZ = deepcopy(z) #deep copy
>>>x[1] = 10
>>>print z
[[1, 10, 3], ['a', 'b', 'c']]
>>>print newZ
[[1, 2, 3], ['a', 'b', 'c']]
```

`string`

The `string` module contains an extensive set of string constants, as well as some string functions. Among the constants are:

`digits`
: The string '012345678'.

`lower case`
: The string `'abcdefghijklmnopqrstuvwxyz'`.

`upper case`
: The string `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Many of the string functions in this module are now methods of type `string`, and are therefor obsolete.

`os.path`

In this module you will find many useful functions for manipulating file paths, such as finding the directory of a path or the base name of a file, to join two paths together, and to check for the existence of a path.

`time`

This module contains functions for getting the current time, and converting

epoch:

the time to various formats. Computers represent time as the number of seconds since the *epoch*. The epoch is the point where time starts, and usually is 1 January 1970. The following can be used to get the current date and time:

```
>>>from time import localtime
>>>print localtime()
(2007, 2, 4, 13, 34, 24, 6, 35, 0)
```

The tuple represents (year, month, day, hour, minute, second, weekday, yearday, DST[2]).

2.5.3 Excercises

1. Implement a function that prints the current date like this: `'5 February 2007'`.
2. Implement a function that prints a given number of random numbers in words, like this:

```
four
one
...
```

[2] Daylight Savings Time.

II. XSI Scripting

3 Basic XSI Scripting

3.1 Introduction

Here we will look at the very basics of scripting XSI: getting hold of scene objects, adjusting their properties, driving parameters using f-curves, using XSI collections, and a little bit about using the tool development interface.

3.2 The XSI SDK

The XSI SDK (Software Development Kit) is a huge body of software that allows you to manipulate XSI in a way that cannot be done through the interface. It consists out of several components, of which the Scripting Command API (Application Programming Interface) is one. Other components can be used to interface to XSI through C++, develop shaders, manipulate the dotXSI file format, or develop custom FX operators; these will not concern us here.

The scripting API allows us to do anything in XSI that can be done through the user interface: we can build scenes, animate models, change shortcut keys, and save files. There are about 1800 commands, 500 classes, 1700 methods, and 1700 data attributes.

3.3 The Tool Development Environment

Although XSI does not provide all the rich features of a proper IDE (Integrated Development Environment, that is, a tool for developing software), it does have some features that greatly simplifies scripting. The Tool Development Environment layout (access from menu under View, Layouts) provides, among others, the following panes:

Scripting panes: This is where you can load, write, and run scripts. A scripting pane does not provide syntax highlighting, but it does perform crude automatic indentation. If you select a command, and press F1, you will be taken to a list of relevant topics in the DSK documentation. You can also get syntax help from the context menu (right-click to access).

It is a good idea to always keep your main script in the first pane, and use one of the other panes as a scratch pad where you can try out short snippets of code before you incorporate them in your main script.

You have to set the language to Python for Python scripts to work To do this, in your script pane, from the menu, select File, Preferences, and select “Python ActiveX Scripting Engine” as your scripting language.

Output pane: Output messages (from scripts), warning messages and error messages are printed in this pane. Your actions in the interface are also logged in this pane, for instance, if you select an object named `'sphere'`, the line

```
Application.SelectObj("sphere", "", 1)
```

is logged. This command appears as a hyperlink to the SDK documentation (to enable this functionality, in your script pane, select File, Preferences, and switch “Show Command hyperlinks” on).

SDK Explorer: This pane provides information about objects and other XSI elements relevant to scripting, such as their types and attribute values.

Netview Pane: This is a built in browser with special XSI capabilities. For examples, if a page contains a link to a script, it can be executed by clicking on the link.

3.4 Scene Objects

3.4.1 Accessing scene objects

There are several ways we can get hold of an XSI object in our scene:

- use its name as a parameter to the function `Application.getValue`;
- use the property `children` of an appropriate object;
- use the property `parent` of an appropriate object;
- query the application for the selected objects, through the property `Application.Selection`; or
- use the return value of the function with which the object was created;

Suppose we have a scene with a sphere (named `'sphere'`) with a child cube (named `'cube'`).

Here are a few examples of how to access the cube:

```
### Accessing an object using its name
cube = Application.GetValue('cube')
Application.LogMessage(sphere)

### Accessing an object using the scene tree
cube = Application.activeSceneRoot.children[2].children[0]
Application.LogMessage(cube)

### Accessing an object through selection
cube = Application.Selection[0] #works only if the cube is selected
Application.LogMessage(cube)
```

Obtaining an object through its name is very convenient, especially for quick and dirty jobs. Because of the script's dependence on this name, this methods is not well-suited for reusable scripts. It is also not a very good method of handling large sets of objects. We will not be using this method.

3.4.2 Selected objects and Picking

The variable `Application.Selection` gives access to the selected objects in the scene.

Set up you scene as follows:

Now run the following script:

```
selection = Application.Selection

for object in selection:
   Application.LogMessage(object)
```

Select some snippets in your scene, and run the snippet. You will see that the snippet prints the names of all the selected objects.

The following piece of code sets the selection to the children of the selected objects:

```
### Changes the selection to the children of selected object
newSelection = []
selection = Application.Selection

for object in selection:
   newSelection += object.children

if len(newSelection) > 0:
   Application.selectObj(newSelection[0])

   for object in newSelection[1:]:
      Application.addToSelection(object)
```

We might want to develop a tool that works on selected objects, but where the objects play different roles. For instance, suppose we want a tool that adds the local position of an object (the “source”) to another object (the “destination”). The `Application.Selection` object will not work, for we cannot predict which object will be the source and which the destination. In such cases, we can invoke a pick session, letting the user pick objects in the order we determine. Here is how we would implement the “add location” script through picking:

```
from win32com.client import constants

srcPicked = Application.PickElement(constants.siObjectFilter,
   'Pick source', 'Pick source')

srcObject = srcPicked[2]

destPicked = Application.PickElement(constants.siObjectFilter,
   'Pick source', 'Pick source')

destObject = destPicked[2]

destObject.PosX = destObject.PosX.Value + destObject.PosX.Value
destObject.PosY = destObject.PosY.Value + destObject.PosY.Value
destObject.PosZ = destObject.PosZ.Value + destObject.PosZ.Value
```

The `Application.PickElement` command takes 7 arguments:

`SelFilter`
: is of enumeration type `siFilter`, for example `siObjectFilter` or `siModelFilter`. A complete list of filters is given in the documentation under **Scripting Reference/Constants/F/FilterConstant**. See below for an explanation of enumeration types.

`LeftMessage`
: Supposedly the status bar message for the left mouse button (I don’t see it).

`MiddleMessage`
: Supposedly the status bar message for the middle mouse button (again, I don’t see it).

`PickedElement`
: This is an output argument, and can be ignored (see below for explanation)

`ButtonPressed`
: An output argument.

`SelRegionMode`
: An integer that specifies the type of selection to perform. The default, 0, uses the current selection mode. Refer to the documentation for the other modes.

`ModifierPressed`
: An output argument.

The three output arguments is where you would put variables that would contain the output had you used VBScript. If you use Python, the output is returned as a tuple instead. The PickElement command has three output arguments, therefor a tuple with three elements is returned, in this order: buttonPressed, modifierPressed, pickedElement. The order can be deduced from the examples in the SDK documentation. As you can see, in this example it does not conform to the order given in the commands parameter list.

enumeration type:

The first argument is a constant. Groups of constants are often given a name by the XSI documentation, in which case the group of constants is an *enumeration type*. Other languages provides explicit support for enumeration types. In Python, however, we merely use this "type" name to determine which constants are valid in a particular situation. You must import `constants` from the `win32.client` module to use the constant variables.

You should always check whether the right button has been pressed in a picking session, and halt execution if it has. Here is an improved version of the script above:

```
from win32com.client import constants

def pickElements():
   srcPicked = Application.PickElement(constants.siObjectFilter,
      'Pick source', 'Pick source')

   if srcPicked[0] == 0: #0 is right mouse button
      return

   srcObject = srcPicked[2]

   destPicked = Application.PickElement(constants.siObjectFilter,
      'Pick source', 'Pick source')

   if destPicked[0] == 0:
      return

   destObject = destPicked[2]

   destObject.PosX = destObject.PosX.Value + destObject.PosX.Value
   destObject.PosY = destObject.PosY.Value + destObject.PosY.Value
   destObject.PosZ = destObject.PosZ.Value + destObject.PosZ.Value

pickElements()
```

3.4.3 Creating Objects

! →

There are many methods with which objects can be added to a scene. They basically work the same, so we will look only at one example: `addGeometry`. Other methods for adding objects can be found in the DSK documentation under **Scripting Reference/Objects/X/X3DObject**.

To add geometry as the child of an object (say, the scene root), we call the `addGeometry` method of that object. The method returns the (Python version of) the newly created object, which we can further manipulate through code. The following example adds a cube to the scene root, and prints its name to the output pane:

```
cube = Application.ActiveSceneRoot.addGeometry('Cube',
   'NurbsSurface', 'MyCube')

Application.LogMessage(cube)
```

The method's first parameter is a string specifying the object's preset; the second parameter is a string specifying the type of geometry. A list of presets and geometry types can be found in the SDK documentation under **Scripting Reference/Other Reference/Primitive Presets**. The final parameter is the name of the new object. XSI automatically adds a counter to the name if it exists.

! →

The following piece of code creates 100 randomly positioned and randomly sized spheres:

```
from random import random

for i in range(100):
   sphere = Application.ActiveSceneRoot.addGeometry('Sphere',
       'NurbsSurface')
   sphere.radius = 2 * random() + 1
   sphere.PosX = random() * 15
   sphere.PosY = random() * 15
   sphere.PosZ = random() * 15
```

3.4.4 Exercises

1. Write a script that will create the following primitives:
   (a) a icosahedron;
   (b) a grid; and
   (c) a null (Hint: use the AddNull method); and
   (d) a spiral (Hint: use AddPrimitive).
2. Implement a function that takes a X3DObject as an argument, and adds a sphere as a child to that object. Then use this function in a script that adds objects to all the selected items.
3. Develop a script that inverts the names of two objects the user picks.
4. The PickPosition command is similar to the PickElement command:

   ```
   buttonPressed, x, y, z = Application.PickPosition (
      'Pick position', 'Pick position')
   ```

   Use it in a scripts that lets the user pick a position, and adds a sphere in that location. Remember to halt execution if the right mouse button has been clicked.

3.5 Object Properties

In the last example, we have accessed some of the sphere's properties to change its size and position. In general, properties are data attributes, and are accessed through the dot notation. Many properties look and behave like they are of primitive types (such as integers or strings). They can be assigned with these values, they can be used as these values in expressions, and they appear as these when printed.

```
sphere = Application.ActivePrimitive.addGeometry('Speher',
   4'NurbsSurface')
sphere.PosX = 5
sphere.PosZ = sphere.PosX + sphere.PosY

Application.LogMessage(sphere.PosZ)
```

However, the following gives an error message:

```
sphere = Application.ActivePrimitive.addGeometry('Sphere',
   'NurbsSurface')
sphere.PosX = 5
sphere.PosX += 5
```

The reason is that this property is *not* in fact a number, but an emulation type that does not support the `+=` operator. To get the actual numeric value, we have to access the `Value` property of `PosX`, like this:

```
sphere = Application.ActivePrimitive.addGeometry('Sphere',
   'NurbsSurface')
sphere.PosX = 5
sphere.PosX.Value += 5
```

The illusion that `PosX` is a number was created intentionally by the developers of the scripting API—to support f-curves and other bells and whistles, `PosX` must be a complex type; letting it emulate a numeric is for our convenience.

3.5.1 Kinematics Object

`Kinematics:`
kinematics state:

Every object has an attribute *Kinematics*. This object has two attributes, each a *kinematics state* object, which holds the local and global kinematics states. They are called `local` and `global`. Among others, these objects have the following parameters:

`PosX, PosY, PosZ`
: The position of this object.

`RotX, RotY, RotZ`
: The rotstion of this object. These can be found under the *Ori* group in the explorer.

`SclX, SclY, SclZ`
: The scale of this object.

`SclrX, SclrY, SclrZ`
: The shear of this object. These can be found under the *SclOri* group in the explorer.

```
sphere = Application.ActiveSceneRoot.addGeometry('Sphere',
   'NurbsSurface')

sphere.Kinematics.Local.PosX = 5
sphere.Kinematics.Global.PosY = 1
sphere.Kinematics.Local.RotX = 90
sphere.Kinematics.Local.SclZ = 2
sphere.Kinematics.Local.SclrX = 45
```

You might be wondering why we have used `sphere.posX` before, and not `sphere.Kinematics.Local.P`
The shorter notation is equivalent to the longer version. It is provided as a shortcut to the script developer. The attributes of the local kinematics state object have been *promoted* to appear as attributes of `X3DObject`. In the SDK documentation

promoted:

! → **Scripting Reference/Shortcut Reference** gives a list of all shortcuts. Note that the global kinematics attributes cannot be accessed using shortcuts.

3.5.2 XSI Collections

The scripting API uses collections extensively. Parameters (of an object), keys (of an f-curve) and selected items are all stored in specialised XSI collections. While some of these collections provide extra methods or attributes, all emulate sequence types, so that you can use familiar notation to access elements and iterate through them.

```
for object in Application.Selection:
   Application.LogMessage(object)

firstObject = Application.Selection[0]
```

Here is a useful code snippet that prints the names of all the attributes of an object:

```
def printParameters(obj)
   for parm in obj.Parameters:
      Application.LogMessage(parm.FullName)
```

3.5.3 F-Curves

f-curve:

A *f-curve* can be added to a property by using the function `addFCurve2`. The function takes a list (or tuple) of alternating frame numbers and values. The following example adds an f-curve to an object's x-position.

```
cube = Application.ActiveSceneRoot.addPrimitive('Cube',
'NurbsSurface')

cube.PosX.addFCurve2([1, 0, 11, 5, 21, 0])
```

If a parameter is already driven by a f-curve, we can use the `source` command of that parameter to manipulate the f-curve using the methods of the class `FCurve`. In the following example, we adjust the times of the keys:

```
cube = Application.ActiveSceneRoot.addPrimitive('Cube',
'NurbsSurface')
cube.PosX.addFCurve2([1, 0, 11, 5, 21, 0])
fCurve = cube.PosX.source

time = 1 #first frame

for key in fCurve.Keys:
   key.Time = time
   time += 20 # next frame will
              # be 20 frames later
```

We can also see what a parameters value will be at a given frame (not necessarily a key-frame) by evaluating the f-curve at that frame using the `Eval` method:

```
cube = Application.ActiveSceneRoot.addPrimitive('Cube',
   'NurbsSurface')
cube.PosX.addFCurve2([1, 0, 11, 5, 21, 0])
fCurve = cube.PosX.source

Application.LogMessage('PosX at frame 4 is ' +
   str(fCurve.Eval(4)))
```

3.5.4 Exercises

1. Implement a function that takes four arguments: an X3DObject and three floating point numbers. The function must set the objects local position to the three given numbers.
2. Implement a function that takes four arguments: an object and three floating point numbers. The function must add a Phong material to the given object, and set the diffuse and ambient color of the object to the given RGB values.
3. Implement a function that returns a list of values that can be used to make an f-curve. The function must take the first frame, the last frame, the step size, the beginning value of the parameter, and the end value of the parameter. For instance,

   ```
   makeFCurve(1, 10, 1, 0, 90)
   ```

   must return the list

   ```
   [1, 0, 2, 10, 3, 20, 4, 30, 5,
      40, 6, 50, 7, 60, 8, 70, 9, 80,
      10, 90]
   ```

4. Modify the function above to take an additional argument. The additional argument is a function, that takes a number as argument, and returns the argument.

   ```
   makeFCurve(1, 10, 1, 0, 90, cos)
   ```

   ```
   [1, 1.0,
      2, -0.83907152907645244, 3, 0.40808206181339196,
      4, 0.15425144988758405, 5, -0.66693806165226188,
      6, 0.96496602849211333, 7, -0.95241298041515632,
      8, 0.63331920308629985, 9, -0.11038724383904756,
      10, -0.44807361612917013]
   ```

5. Implement a function that takes four arguments: an X3DObject, an integer (`childrenCount`), and two floats (`radius`, `minDistance`). The function must add `childrenCount` spheres to the object, with radius `radius` and at a random location at least `minDistance` from the parent, but not further than twice `minDistance`.
6. Modify the function above to take another integer argument `depth`. If this argument is 0 or less, the function should return without doing anything. Otherwise the function must call itself again, but pass each child as parameter, with half the radius and minimum distance, and one less depth.
7. Modify the function above to add a f-curve to the parent object so that the object revolves around one of its axes - one revolution in 100 frames (from 1 to 101).
8. Modify the function above to call your coloring function implemented earlier.
9. Write a script that adds a single sphere to the scene root, color it (using your coloring function), animate it (add a f-curve to it), and call the function above on it. Play around with the paramters until you get a decent effect.
10. Implement a function that takes two X3DObjects, and returns a duplicate of the first, with its local position parameters set to halfway between the two objects. (Hint: use the `Application.Duplicate` command). Now write a script that calls this function on two user picked (not selected!) objects.
11. Implement a function similar to the one above, but that will work if the positions of the objects have been animated using f-curves. Use the first frame to do the calculation.
12. Implement a function similar to the one above, except that it should add a f-curve to the object so that the object's position is always the average of the other two objects'.

3.6 Application Information

The global object Application provides information about the application's status. Here are a few of its methods:

`ActiveProject`
: The active project. Changing this value changes the active project.

`ActiveSceneRoot`
: Gives the active scene's root model.

`ActiveToolName`
: Gives the name of the currently active tool, i.e. one of `RotationTool`, `ScalingTool` or `TranslateTool`. If the tool is not recognised, an empty string is given.

`Commands`
: A collection of all the available commands, including built-in and custom commands.

`Preferences`
: An object of type `Preferneces` that can be used to manage XSI preferences through scripting.

`Selection`
: A collection of selected objects.

3.7 SDK Documentation

The SDK documentation can be daunting.

Because XSI supports four scripting languages, the documentation use terms for programming concepts that sometimes differ from those in Python. Here is a list of such terms, with their Python equivalents:

Array: Tuple or list. In the case of a return value, it is always a tuple.

Double: Float.

Long: Integer.

Property: Data attribute.

Variant: used to mean "any type". This does not actually mean that a method or function can take any type, just that it is not specified. This is not really important for Python programs, since the type of arguments and return values are *never* specified.

return value:

The XSI documentation uses the term *return value* to describe the *type* of a data attribute. This usage can be confusing, since data attributes do not return values.

scripting syntax:

The *scripting syntax* under a method entry shows how a method should be used, for example:

```
Application.LogMessage (Message, [Severity])
```

shows that `Application.LogMessage` can take two arguments. Arguments in square brackets are optional.

enumeration types:

Enumeration types are groups of named constants, usually strings or integers. For instance, the second argument of `Application.LogMessage` is of pseudo-type `siSeverity`, which contains among others the constants `siFatal` and `siError`. A constant can be used like any other variable.

command:

An XSI *command* is simply a method of the `Application` object. In other languages, commands can be used directly, for example, in VBScript we can use the following:

```
LogMessage "hello"
```

but in Python, we need to call `LogMessage` on `Application`:

```
Application.LogMessage("hello")
```

3.7.1 Useful Pages

The following is short list of useful pages. The path in brackets refers to the table of contents, for example *Answers to Basic Questions* can be found under *Scripting Reference*, under *What's Available*.

! → **Answers to Basic Questions:** (**Scripting Reference/What's Available/**) This section gives a list of links to documents describing some basic concepts of scripting for XSI, including finding the type of an object, the difference between owners and parents, and using collections. You should read all these!

! → **Using shortcuts:** (**Plug-in Developer's Guide/Working with Parameters/ Accessing Parameter Values/**) Provides information about shortcuts.

! → **Issues with Python:** (**Script Development**) Lists and addresses issues specific to Python.

! → **Global (Intrinsic) Objects:** (**Script Development**) Lists the available global objects (such as `Application`) and what they are used for.

4 XSI Scripting

4.1 Introduction

In this section we will look at creating and manipulating materials, cameras and lights through scripting. You will also be introduced to scripted operators—a powerful method of scripting the behaviour of objects.

4.2 More XSI Objects

4.2.1 Materials

Geometry that can have materials applied to them have the method `AddMaterial` with which this can be done. The method takes three arguments: the first is a string denoting the preset (such as `'Phong'`, `'Lambert'`), the second is a *branch flag*, a boolean indicating whether the material should be applied to all nodes in a branch, and the last is the name of the material. The method returns the material object.

branch flag:

The following example creates a sphere, and adds a simple Phong material to it. Note that the material is stored in a variable so that we can alter its other properties.

```
root = Application.ActiveSceneRoot
sphere = root.addGeometry('Sphere', 'NurbsSurface', 'redSphere')
material sphere.AddMaterial('Phong', false, 'redPhong')
```

After we have added the material, we would like to change the properties. These properties are located deep in the object hierarchy, and we will use variables and a helper function to reduce the amount of typing:

```
def setRGB(materialProperty, red, blue, green):
   materialProperty.red = red
   materialProperty.green = green
   materialProperty.blue = blue

root = Application.ActiveSceneRoot
sphere = root.addGeometry('Sphere', 'NurbsSurface', 'redSphere')
materialLib = sphere.AddMaterial('Phong', False, 'redPhong')
phong = materialLib.Surface.NestedObjects('redPhong')

ambient = phong.ambient
setRGB(ambient, 1.0, 0.0, 0.0)

diffuse = phong.diffuse
setRGB(diffuse, 1.0, 0.0, 0.0)

specular = phong.specular
setRGB(specular, 1.0, 1.0, 1.0)
```

Notice that the actual Phong is a element of `NestedObjects`, which is an property of the `Surface` object. The `NestedObjects` is a collection. Like lists, we can use integer indices to access elements. However, this is not always a safe approach: in cases where we cannot control an object's position in a collection, we

dictionary:

cannot predict with complete accuracy where that object will lie. For this reason the `NestedObjects` collection emulates a dictionary. A *dictionary* is a collection that can be indexed using arbitrary objects, not just integers. In this case, the collection is indexed with strings—the (XSI) names of the objects in the collection. Other XSI collections also emulate dictionaries in this way, allowing us not to be concerned about an object's position in a collection.

4.2.2 Lights

Lights are added to a scene much like geometry is, except that we use the `AddLight` method. This method takes three arguments: the first is the name of the preset we want (eg. `'Point'` or `'Neon'`); the second is a boolean indicating whether an interest should be added; the final argument is the XSI name of our light.

The following example sets up three lights around a sphere:

```
def setPos(obj, x, y, z):
   obj.posX = x
   obj.posY = y
   obj.posZ = z

def setLightIntensity(light, intensity):
   light.LightShader.NestedObjects('soft_light').intensity = intensity

def addThreePointLights(root, x, y, z, r):
   light = root.AddLight('Point', False, 'MainLight')
   setPos(light, x + r, y + r, z + r)
   setLightIntensity(light, 1.0)

   light = root.AddLight('Point', False, 'AmbientLight')
   setPos(light, x - r, y - r, z + r)
   setLightIntensity(light, 0.2)

   light.SpecularContribution = False

   light = root.AddLight('Point', False, 'RimLight')
   setPos(light, x-r, y + 1.5*r, z-r)
   light.SpecularContribution = False

root = Application.ActiveSceneRoot
addThreePointLights(root, 0, 0, 0, 5)
root.addGeometry('sphere', 'NurbsSurface')
```

Note that we can access the `SpecularContribution` property of a light directly (through a short-cut), but to set the intensity we have to navigate through various objects to get to it. Notice again how we used a string to access the right element in the `NestedObjects` property. To see which objects we need to access for a certain property, we have to use the SDK explorer tree. It can be tricky, but you will soon get used to it.

4.2.3 Cameras

Since cameras are handled similarly to other XSI objects, we won't say much about them here. The example below is a Python translation of the example that comes with the XSI SDK documentation. This example contains a few things (not specific to cameras) that we have not mentioned before; they are explained below.

```
# Python example demonstrating how to access all
```

```
# Cameras in the scene.

from win32com.client import constants

def printCameraInfo(camera, i):
   print '--------------Camera' + str(i) + '---------------'
   print 'Camera Name: ' + camera.Name
   print 'Camera Root: ' + camera.Parent.Name
   print 'Camera Primitive: ' + camera.ActivePrimitive.Name

   #The Interest a 'sibling' of the Camera so
   #we find it via the parent
   childrenOfParent = camera.Parent.Children

   interest = None

   for obj in childrenOfParent:
       if obj.type == 'CameraInterest':
          interest = obj
          break

   print 'Camera Interest: ' + interest.Name

   #Many aspects of the camera can be manipulated by reading
   #and writing the parameters on these various objects

   if camera.ActivePrimitive.Parameters('proj').Value == 1:
       print 'Projection: Perspective'
   else:
       print 'Projection: Orthographic'

   print 'Field of View: ' + str(camera.fov.Value)

   #Although these parameters actual belong
   #to the local kinematic property,
   #these shortforms are available to python
   #to make it even more convenient

   print 'Interest Pos(local): (' \
      + str(interest.PosX.Value) + \
      ',' + str(interest.PosY.Value) + \
      ',' + str(interest.posZ.Value) + ')'

   globalKine = interest.Kinematics.Global
   print 'Interest Pos(global): (' \
      + str(globalKine.PosX.Value) + \
      ',' + str(globalKine.posY.Value) + \
      ',' + str(globalKine.posZ.Value) + ')'

#Create a sample scene with different cameras
Application.NewScene('', False)

root = Application.ActiveSceneRoot

root.AddCamera('Default_Camera')
root.AddCamera('Telephoto', 'Telephoto')
```

```
#Put the orthographic camera inside a model
model = root.AddModel()
model.AddCamera('Orthographic', 'Ortho')

#Hide one of the cameras underneath a cone
cone = root.AddGeometry('Cone', 'MeshSurface')
cone.AddCamera('Wide_Angle', 'WideAngle')

root.AddCamera('Wide_Angle', 'WideAngle')

#Perform a recursive search for all the cameras in the scene
cameras = root.FindChildren('', constants.siCameraFilter )

#Print out some information about the Cameras in the scene
print 'The scene contains ' + str(len(cameras)) + ' cameras'

for i, camera in enumerate(cameras):
   printCameraInfo(camera, i)
```

In the example above, you might be temped to replace

```
... str(interest.PosX.value)
```

with something shorter:

```
... str(interest.PosX)
```

Unfortunately, the code above will not behave as expected. Remember that `PosX` parameter only *emulates* a numeric type, and only in certain contexts. In the above context, `PosX` behaves more like the complicated type it really is, and the `str` function coverts it to the parameter *name*, which is `CameraInterest.kine.global.posz`.

If you code in VBScript, you can access XSI constants directly. If you use Python, however, you must import the `constants` object (from `win32com.client`), and access all constants through this object. XSI constants are usually prefixed with `si`, which make them easy to spot in the documentation and code snippets written in other languages.

4.2.4 Exercises

1. Implement a function that will add a Phong to an object that will make it transparent. The amount of transparency must be a parameter of the function.
2. Implement a function that will invert an object's ambient and diffuse colors. Your function must do this to *all* materials in the object's material library. Assume they are all Phongs.
3. Modify the function above to work with other material types as well. For each material in an object's material library, check which type of material it is, and change the appropriate values.
4. Implement a function that will replace all the Phong materials with suitable toon shaders. (Use the diffuse color for the toon shader's color).
5. Implement a function that will replace 1 light with three copies. Give the copies a slight random offset, and change their intensities to a third of the original light. Two of the copies must have specular and shadows disabled.
6. Implement a function that will find all the lights in an object, and make them slightly dimmer. (Hint: use the `FindChildren` method).
7. Implement a function that will create a strobe light (i.e. it flashes on and off). The number of frams between flashes must be a parameter of this function.

8. Implement a function that adds a camera to an object, and animate it in a circle around the root.
9. Implement a function that allows a user to select a point in space, and an object of interest. Add a camera at the point in space, and parent the camera's interest to the selected object.

4.3 Scripted Operators

scripted operator:

A *scripted operator* is a function that controls a parameter of an object. This function might take parameters of other objects as its arguments. It is called automatically by XSI, so that the parameter you are trying to control is always up to date.

Because scripted operators are called automatically, they must be of a specific form. It always takes at least two arguments. The first is a *context* parameter, giving you some information about where the function was called from. The second parameter is the output argument. This parameter's `Value` attribute will be the value of your parameter; you should assign this with whichever value you want your parameter to be. If you want your scripted operator to take more arguments, you have to add them explicitly with the *New...* button. The explorer tree will pop up, and allow you to select the parameters you want directly. This alone makes scripted operators easier to use than normal scripts. Note that the parameters do not emulate numeric types in this context—you have to extract the Value explicitly using the `Value` attribute.

The following example is a scripted operator on a cylinder with unit radius. If the cylinder is rotated 90 degrees about the x-axis and parented to a cube, it will roll with the cube as it is translated in the x direction. Note that the first line is provided to emphasise that we are working with a Python function; it is generated automatically, and appears just above the editing region.

```
def Update(In_UpdateContext, Out, Inposx):
   Out.Value = -Inposx.Value * 360.0 / (2 * 3.14)
```

4.3.1 Exercises

1. Implement a scripted operator to control the intensity of a light. The intensity of the light must be proportional to the squared distance of the light to an object (any object you choose). (Hint: the squared distance between two points is given by $(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2$.
2. Implement three scripted operators that will make an object (the predator) move in the direction of another object (the prey). Animate the prey, and see what the predator does! (Hint: Increment PosX if the prey's PosX is larger than the predators PosX, otherwise decrement it. Do the same for the PosY and PosZ).

5 Good Programming Practices

5.1 Introduction

There are many ways in which to write a script that does some specific task; this immediately implies some are better than others. Here we will look at good programming practices, and why they are good.

5.2 What makes good code?

What makes good code? Code is good if:

- it works;
- it is simple to read and understand;
- it is simple to write;
- it can be re-used;
- it is fast;
- it uses little memory.

The first criterion is obviously the most important, and in this document we will take it for granted that whatever code we write, it must perform the task it was designed to do. Nothing is just black and white: later we will see how usability and portability affects how well a program works, and that these factors must also be weighed with the factors presented here.

The next to points go hand-in-hand. You will frequently find a new use for old code. If it was badly written, it will be a hard task to figure out what everything does and where you have to modify it to work in the new context. Even while writing code, it is very easy to forget what values are stored in a particular variable, or what a certain function does. So it is harder to write on programs that read difficult.

Python provides many facilities to help in this respect, and they fall into two categories: naming and organisation. We will address these in detail in subsequent sections.

Code that is reusable is more valuable than code that is not, since it saves the programmer from starting from scratch. There are different levels of re-use.

The crudest form (and also the one most frequently employed) is to simply copy code from somewhere, and modify it to work in the new program. Readability plays a major role in this kind of re-use.

A more sophisticated form of re-usable code is a library of related functions and classes that serve as building blocks for many different programs *without modification*. This level of reusability can be obtained through proper organisation of code.

Some programs are so re-usable that it is not necessary to write certain additional programs *at all*. This feat is accomplished by proper user requirements analysis and proper program design.

As mentioned before, naming and code organisation will be covered in detail in following sections. Program design is a big topic in itself, and will not be covered here.

Speed and memory consumption—the two sides of efficiency—are of lessor importance, as long as you watch out that something that can be processed in seconds does not take minutes or hours, or that a program that really only needs a megabyte of memory uses one gigabyte. We will only skim over some principles in a following section.

You will see that these things often work against each other or even themselves. Efficient code is often cryptic; descriptive names are longer and can hinder writing of code; extreme organisation can lead to huge libraries of unnavigatable code; a reusable function might be difficult to use because of the amount of parameters it takes. You have to weigh up the importance of various factors, and find the balance that will reflect this balance.

5.3 Naming

5.3.1 Variables

Variables should be named using *nouns*: `sphere`, `mainCamera`, `rimLight`.

Using single letters for variable names are not acceptable, except in the following cases:

- When used to denote RGB or RGBA values.

```
setRGB(colorProperty, r, g, b):
   colorProperty.red = r
   colorProperty.green = g
   colorProperty.blue = b
```

- When used to denote x, y, z-coordinates.

```
setLocalTranslation(obj, x, y, x):
   obj.PosX = x
   obj.PosY = y
   obj.PosZ = z
```

- When used to denote loop counters, in which case i, j and k are used.

```
for i in range(0, 10):
   for j in range(0, 10):
      for k in range(1, 10):
         root.AddSphere(i, j, k)
```

- When used to denote time (the frame number), in which case t is used:

```
getLocalTransform(obj, t):
   x = obj.PosX.Value(t)
   y = obj.PosY.Value(t)
   z = obj.PosZ.Value(t)

   return (x, y, z)
```

flags:

Variables that hold boolean values are usually prefixed with `is-` or `has-`: `isNurbsSurface`, `hasChildren`. These variables are sometimes called *flags*.

List of objects are usually named by plural forms: `cameras`, `children`. The singular form is used for iterating over the list:

```
for camera in cameras:
   #do something with camera

for child in children:
#do something with child
```

A variable that denotes the number of some item in a data structure (such as a list) is postfixed with `-Count`:

```
cameraCount = len(cameras)
childCount = len(children)
```

It is customary for XSI script writers to prefix objects with an `o-` (as can be seen in the XSI examples): `oCamera`, `oSphere`. I think that this usage does not povide any useful information, and makes code less readable. Even so, you might decide to adopt this practice to be consistent with other code writers.

5.3.2 Functions

Functions should be named by using *verbs*: `addSphere`, `animateObject`, `calculateLength`. If you find it difficult to give a function a good name, it is often a sign that the function does too many things, and should be split into different functions.

accessor: getter:

A function that returns a property of an object is called an *accessor* or *getter*. Accessors' names are usually prefixed with `get-`: `getRed`, `getLastKeyValue`, `getIntensity`. If the property is a boolean value, the prefix `is-` is used instead: `isNurbSurface`, `isAnimated`, `isValidPosition`.

mutator: setter:

A function that change some properties of an object is called a *mutator* or *setter*. Such a function's name is usually prefixed with `{set-}`: `setSize`, `setRGB`, `setLocalTranslation`.

The prefix `init-` is used for initialisers: functions that sets the properties of an object for the first time: `initMainLight`, `initCharacterPose`.

The prefix `copy-`, `clone-`, or `duplicate-` is used for functions that duplicate objects: `duplicateSphere`, `duplicateModel`. Do not use more than one of these prefixes in a program.

5.4 Organisation

5.4.1 Functions

- A function should perform a single, self-contained task.

```
### Bad
def setPositionAndAddSphere(obj, x, y, z):
   obj.PosX = x
   obj.PosY = y
   obj.PosZ = z

   obj.AddGeometry('Sphere', 'NurbsSurface')

### Better
def setPosition(obj, x, y, z):
   obj.PosX = x
   obj.PosY = y
   obj.PosZ = z
```

```
def addSphere(obj):
   obj.AddGeometry('Sphere', 'NurbsSurface')
```

The first function is less re-usable, and is awkward to conceptualise and hence will make code less readable.

- A function should obtain all the objects it must work on as parameters. Compare the following:

```
### Bad
def addSphere():
   Applicaion.ActiveSceneRoot.AddGeometry('Sphere', 'NurbsSurface')

### Better
def addSphere(obj):
   obj.AddGeometry('Sphere', 'NurbsSurface')
```

The second function is better, because it can add a sphere to an arbitrary object, not just the scene root. (The scene root can be passed as an argument to the second function if the sphere should be added to the scene root).

5.4.2 Scripts

- Always use functions to do actual work. The main body of your script should extract the necesary scene objects (for example, the scene root), and call an appropriate function.
- Separate functions that might be useful for other scripts from functions that are very specific to the task at hand. Put the re-usable functions in a library (a separate script).
- A script should be organised as follows:
  1. A comment that describes what the script does and how it should be used. Remember to include your name as the author.
  2. A block of import statements.
  3. A block of functions. Put related functions near each other.
  4. The main script body.

5.5 More Techniques for Creating Re-usable Code

- Avoid making your scripts dependent on *names*. In some cases using names is unavoidable; if you name objects in your script, make sure that your script works even if XSI append a number to your name. For example:

```
sphereName = 'mySphere'
sphere = obj.AddGeometry('Sphere', 'NurbsSurface', sphereName)
sphereName = sphere.Name #for in case XSI renamed my sphere

... #do something with sphereName
```

- Avoid relying on the order of items in a list. Search through a list for objects using properties, types, or in rare cases, names.

magic number:

- Avoid using *magic numbers*, i.e. a literal numeric values such as `12` or `3.14`. Rather use named variables, so that you have a clue where the number comes from:

```
### Bad
for i in range(12):
   createConstellation(i)
```

```
### Better
CONSTELATION_COUNT = 12

for i in range(CONSTELATION_COUNT):
   createConstellation(i)
```

5.6 Efficiency

Normally, efficiency is not a concern for an XSI scriptwriter. There are a few things that you should keep in mind, though:

- The object model is generally faster than commands.
- Loop bodies should contain only necessary statements. Consider for example:

```
### Bad
for i in range(1000):
   sphere = obj.AddGeometry('Sphere', 'NurbsSurface')
   rotx = obj.RotX
   sphere.PosX = rotx * i

### Better
rotx = obj.RotX

for i in range(1000):
   sphere = obj.AddGeometry('Sphere', 'NurbsSurface')
   sphere.PosX = rotx * i
```

  In the last example, the statement `rotx = obj.RotX` does not need to bee in the loop, so it has been put outside the loop.
- Use of functions reduces memory consumption if you make proper use of local variables. The memory that a variables uses is freed when the variable goes out of scope.
- It is possible to animate properties by manipulating the `PlayControl` object. This is extremely slow; rather manipulate f-curves directly.

5.7 XSILibrary

This section is one major exercise. The functions that are implemented in this module are helper functions that you should find useful in later work.

```
getSiblings(obj)

setLocalTranslation(obj, x, y, z)

setLocalRotation(obj, x, y, z)

setLocalScale(obj, x, y, x)

getLocalTranslation(obj)

getLocalRotation(obj)
```

```
getLocalScale(obj)

setRGB(property, r, g, b)

setAlpha(property, alpha)

setRGBA(property, r, g, b, a)

getRGB(property)

getAlpha(property)

getRGBA(property)

getMaterial(materialLib, index)

getMaterial(materialLib, name)

getMaterials(materialLib)

setLightIntensity(light, intensity)

makeFCurve(begin, end, step, values)

makeFCurve(begin, end, step, function)

selectAll()
```

Index