Phix

A Rudimentary Unauthoritative Introduction for Beginners

Second Draft, January 14, 2023

Please comment, and help fixing typos, factual errors, crucial omissions, confusion, etc.

The Phix Website:
http://phix.x10.mx/

Table of Contents

About …

About Phix

Phix was created and is maintained by a single person, Pete Lomax. Phix is based on Euphoria, which has been around since the 1990s — the differences between the Phix and Euphoria languages are minimal, but the Phix interpreter/compiler is completely independent. This is not the place to go into technical details, but (as an entirely subjective statement), Phix is more pleasant to work with.

(If you *are* interested in technical details: "Euphoria … is written in C with some parts self hosted and transpiled to C, which should theoretically make it easier to port. Phix has been around quite a bit less, since just before 2010, is a complete rewrite, and is entirely self-hosted/directly generates executables. Both require an existing working version to build, but that can be done on a completely different operating system. The chicken and egg problem was/is solved in Euphoria via C, and was/**is** no longer needed in Phix via FASM.")

The Phix website is http://phix.x10.mx/
The current version of Phix is 1.0.1

A place to get help with Phix and Euphoria problems is https://openeuphoria.org/forum/index.wc — even if most of the discussion there is on a technical level where I don't even understand what they are talking about, I am sure you will receive answers to more mundane questions there, too.

About this Introduction

First a disclaimer: I am in no way involved in the development of Phix, Euphoria, or any other computer language. I'm only a user, who loves Phix. I've started using Euphoria back when it was neat, simple, and fascinatingly concise. There was a lot it didn't have, but you could easily write it yourself, when you needed it. Meanwhile, the scope of the language has multiplied, which for most users is a good thing. I still tend to ignore most of what hasn't been there 20 years ago — you can use only a small part of Phix's features, or explore all the depths and hidden corners of this amazing language, whatever works for you best.

This introduction is not a tutorial or a manual. It does not attempt to teach you Phix or how to write software, it is only meant to let you take the very first steps, and to give you an idea of how Phix is different from other languages. The examples here hardly go beyond various aspects of "Hello World." Everything else you need, you'll find in the excellent and detailed documentation that Phix comes with. This introduction also assumes that you have at least a little experience with writing software. Phix is well suited for beginners, but if you're new to programming, you need a better teacher than I am — yourself, for instance.

It is of course possible to write GUI programs with Phix, but you'll need someone else to tell you how. This text is only about programs that run in text mode in the Windows console (or the PowerShell, but I still prefer the console).

Robert Schaechter
http://hypatia-rpn.net
January 2023

Hello World

When you have installed Phix, you can create an empty folder anywhere you like, to start writing your first Phix program.

All you need is a text editor, preferably one that is better than Windows Notepad. There is one that comes with Phix, Edita, but at least at the beginning I'd suggest you use an editor that you are familiar with. And even Notepad can take you through the first steps.

And, you need to know how to open a console window in the folder you want to use (for instance, open that folder in Windows File Explorer, and type `cmd` in the address bar). You can also use PowerShell instead of the console, but I'm used to the console and prefer it — use PowerShell at your own peril!

The default file extension for the main file of a Phix program is .exw. Create a file hello.exw, write the following line, and save it (you can keep the file open):

```
    puts(1, "Hello World")
```

This is a complete and valid Phix program. Admittedly, "`puts(1, …`" doesn't look pretty, we'll deal with that, but this is the command in Phix to write a text to the screen: output a string to device #1.

When you have saved the file, and opened the console window, type this at the Windows command prompt:

```
    C:\...>p hello
```

The Phix interpreter/compiler starts, reads hello.exw, runs the program it contains, and returns to the command line. On the screen you will see:

```
    Hello World
    C:\...>
```

As easily, you can turn your little Phix program into a stand-alone executable file — all you need to do is add the `-c` option after `p`:

```
    C:\...>p -c hello
```

Your program will run the same as without the -c option, but in addition, Phix will compile your program: you will find a file hello.exe next to hello.exw. This executable will run on any Windows computer — when you run it from the command line, it will again write `Hello World` to the screen. (If you run it from the Windows desktop, it will do the same, but you won't see it, because it will open a console window, write the text, and immediately close the window again, probaby before you've even noticed that something has happened. To avoid this, your program will need to wait for some user interaction before it terminates.)

Your hello.exe file will have a size of somewhat less than 280 KB, which is rather a lot for such a small program, but the good news is, the executables grow only slowly, as your programs get larger and more complex. And, even with much larger and more complex programs, Phix loads and compiles your program amazingly fast.

I strongly advise you that you actually perform these two steps, so we know that Phix is working correctly on your system. Everything from now on you can try out, or you can consider it a thought experiment.

Let’s add a second line to our “Hello” text:

```
puts(1, "Hello World")
puts(1, "It’s me!")
```

When you run this program, you’ll be disappointed, because what you’ll get is

```
Hello WorldIt’s me!
```

Phix’s *puts* command does not end your text with a line break (unless it’s at the end of the program), you'll have to do this yourself — a line break, as part of a text in double quotation marks, is \n:

```
puts(1, "Hello World\n")
puts(1, "It’s me!")
```

Honestly, I find this a bit cumbersome, but it gives us the opportunity to see how easily we can write our own routines, to make things work as we want them.

Edit hello.exw, so that it contains these lines — when you run this program, it will write the same two lines to the screen, but we have replaced Phix’s command with our own, which is easier to use:

```
procedure disp(string text) -- display a line of text on the screen
  puts(1, text & "\n")
end procedure
-- program starts here:
disp("Hello World")
disp("It’s me!")
```

Six simple lines, but they give us a lot to talk about.

First, everything that begins with `--` is a comment and gets completely ignored by Phix, whether it’s a line of its own or comes after some program code in the same line.

Then, *procedure*. Phix knows two kinds of subroutines: functions and procedures. The main difference between them is that a function returns a value (or a sequence of values — what that means, and how functions are referred to, we’ll discuss later). Here we do not need anything to be returned, so we use a procedure. (Strictly speaking, there is a third kind of subroutine, *type*, but we will not discuss it here.)

Before we look into the details of our three-line procedure, an important note: Phix wants to see subroutines before they are called — if not, you get reprimanded with a warning message. There are three ways to deal with this:

a) Let the main program come at the end of your code, following the subroutines, and put each subroutine after any other routine which it calls. The executable part of the main program starts with the first line that is neither a variable declaration (we’ll come to these later), nor is located between a *function* or *procedure* declaration and the corresponding *end function* or *end procedure* statement. This is what we’ve done in our example, and it’s what I usually do.

b) For each subroutine, you can use a *forward declaration:* a line in the code that tells Phix that it will find a subroutine of this name, and with these parameters, later in the code. Only this single line has to precede the calls to that function or procedure. Using a forward declaration, our program would look like this:

```
forward procedure disp(string text)
disp("Hello World")
disp("It’s me!")
procedure disp(string text) -- display a line of text on the screen
  puts(1, text & "\n")
end procedure
```

Forward declarations allow two subroutines to call each other. A function or procedure can always call itself, though.

c) You can put functions and procedures in a different file (or several files) — this is particularly useful with large programs. Let us do this with our little “disp” routine. Remove the three lines of the procedure from the hello.exw file, and put them into a file my_routines.e — the default file extension is .e, the name, of course, is whatever you want to name it.

We now have two files (omitting the comments):

The main file hello.exw

```
disp("Hello World")
disp("It’s me!")
```

and my_routines.e

```
procedure disp(string text)
  puts(1, text & "\n")
end procedure
```

But to make this work, we have to make two additions: in the main program, we have to tell Phix to include the file that has the procedure we want to use, and in the included file, we have to make the procedure *global* — that is, making it visible outside of its own file. The two files are now:

hello.exw

```
include my_routines.e
disp("Hello World")
disp("It’s me!")
```

my_routines.e

```
global procedure disp(string text)
  puts(1, text & "\n")
end procedure
```

You may have good reasons to put functions and procedures in the main .exw file, but you can as well relegate them to their own .e file(s) — this allows you to easily use your own routines in several of your projects. Note that the include statement must precede calls to the routines contained in the included file.

Let's look at our procedure declaration now.

```
    global procedure disp(string text)
```

`global` is needed to let you call the procedure from a different file.

`procedure` as we've already said defines a procedure, `function` would define a function.

`disp ...` when naming your procedure or function, take care that the name isn't already used by Phix. One way to do this is by selecting the "Index" tab in the Phix documentation, and enter the prospective name of your procedure or function (this applies to variable names, too). Another way is to write your Phix code using Phix's Edita editor, which indicates Phix keywords with different colors. (Personally I prefer to use the editor I'm used to, though. Other professional editors may be taught to recognize Phix syntax, but that is yet to happen.)

`( ...` a procedure or function can have one agument, or several arguments, or no argument at all, but even without any argument the parentheses are required, both in the definition and in the calling statement. The number of arguments must match, but there is a way to work around this — about that, later.

`string ...` one of Phix's five types of variables. About those, soon!

`text ...` the name you give to the variable that receives its value from the calling statement. This variable exists only within the procedure or function. Variables are passed by value, not by reference, which means that if you change the variable in the subroutine, the variable in the calling routine will remain unaffected. (Variables can be shared across main program and subroutines, though.)

And now it is time to talk about Phix's five variable types, before we return to our little procedure.

Variable Types

All variables in Phix (with the exception of *for … end for* loop counter variables) need to be declared before they can be used — the declaration assigns one of Phix's five types to them.

The data types are *integer*, *atom*, *string*, *sequence*, and *object*. And, really, more are not needed. (You can define your own types, but these do not add functionality, they only put restrictions on permitted values.)

Actually, you could even say there are fewer than five types: integer is a special case of atom, a string is a special case of a sequence, and an object is either an atom or a sequence. In my own programs (do not let this guide you) I only use integer, atom and object. But, let's look at all five:

An *atom* is a single number — any kind of number.

An *integer* is an atom that, well, happens to be an integer. Or rather, an atom that is only *allowed* to be an integer.

```
atom a
a = 5          -- now variable a is an integer
a = a / 2      -- now it isn't
a = a * 4      -- now it's an integer again
```

When your atom variable is an integer, you can use it as an integer, whenever an integer is required. For instance, in a *for … end for* loop, the initial value, loop limit and increment of the loop variable must be integers. `for i = 1 to a do ...` works when a is an atom, with currently an integer value. If the value of a happens not to be an integer, the program terminates with an error message.

Defining a variable as integer keeps it from being assigned a non-integer value by mistake — this is done the hard way, by making the program crash when it happens. But, the crash comes with a detailed explanation, pinpoints the location where something goes wrong, and may avoid errors from ocurring that might be much harder to trace. (This, of course, is meant to happen during development and debugging — in the finished program, type mismatches should not occur.)

A similar relationship exists between sequences and strings, and between objects and all other types, but this requires us to understand sequences — they are what makes Phix unique among all programming languages (apart from Euphoria, of course). But before we go there, here is another important fact:

To Phix, any integer between 0 and 255 is not only a number, but also an ASCII character. Or, in other words, any ASCII character is also an integer. The number 65 is the upper case A. The upper case A is the number 65. It's what you do with it, that determines whether it serves as a character or a number. This may sound a bit like quantum physics, but, once understood, it will not cause you any problems — just the contrary, it actually simplifies things.

But now, finally, *sequences*. Where other languages have arrays, Phix has sequences, and this is where Phix's magic resides. It is also where Phix's pitfalls wait for the uninitiated, but you'll soon be able to avoid them.

The simple form of a sequence is what in other languages you'd call a one-dimensional array: a numbered series of values — in the terminology of Phix, of atoms. These atoms can be integers, and if these integers are all in the range of 0 to 255, they can be thought of as ASCII characters, if you want to think of them that way. (Phix can also handle Unicode, but we won't discuss this here.)

If you explicitly want a sequence variable not to have more than one dimension, and to contain only values 0 to 255, then you can declare it as string. As with integer vs. atom, this only serves to prevent the variable from causing troubles by erroneously becoming what it isn't supposed to be.

Let's take a little detour here, to talk about scope — this will be rather short and simple. Any variable that you declare is, from that point on, available to all the code within the file in which you have declared it — also to code within functions and procedures. Any variable that you declare within a function or procedure is limited to that function or procedure. Any variable that you want to be available to code in different files, you have to declare as global, by writing global before its type. If you declare a variable within a loop or if statement, it remains local to that part of the code.

But now, before we get to the really interesting part where we'll see what sequences are really capable of, let's stay with the basics of the simple one-dimensional-array kind of sequence for a while — here is also where we'll learn about the pitfalls.

The atom/sequence Pitfall

The elements of a sequence have consecutive numbers, the index, always starting with 1. Indices are written in square brackets. Let's write this little program:

```
    sequence s
    s = "Hello world"
    puts(1, s[7] & "\n")
```

This will write the letter w to the screen — the 7th element of the sequence "Hello world".

Instead of seeing a letter we can see a number, by using *print* instead of *puts*.

```
    print(1, s[7])
    puts(1, "\n")
```

Now we get 119 — the numerical value of the 7th element of our sequence. (The second line adds a line break, `print(1, s[7] & "\n")` wouldn't work. "\n" in the source code is converted to the ASCII "line feed" character 10 at compile time. For Windows, the *puts* command adds the "carriage return" character 13, but the *print* command converts 10 back into the \n escape sequence.)

Had we declared our variable s as a string, the result would be exactly the same: w in the first case, 119 in the second. In Phix, characters *are* numbers.

Let's replace the lower case w with an upper case W — its ASCII code is 87:

```
    sequence s
    s = "Hello world"
    s[7] = 87
    puts(1, s & "\n")
```

and we get *Hello World*. Knowing that the ASCII codes of upper case characters A–Z are those of the corresponding lower case characters minus 32, we could also have written

```
s[7] = s[7] - 32
```

But, let's try not to use the ASCII code, but the character W:

```
sequence s
s = "Hello world"
s[7] = "W"
puts(1, s & "\n")
```

and what do we get? An error in line 4: "sequence found in character string". But why?

Let's try with declaring s as a string, which is what it is supposed to be, after all.

```
string s
s = "Hello world"
s[7] = "W"
puts(1, s & "\n")
```

Now we get a "type check failure" error message for line 3, with the addition:

```
s is {72,101,108,108,111,32,"W",111,114,108,100}
```

About those curly brackets, later.

So, obviously it is ok to assign the value 87 to the 7th element of our sequence, but if we assign it the character "W", it doesn't work. Why not?

The answer comes in three parts:

1. A sequence (and thus also a string) can have any length, and that includes the length zero, and the length 1. But, a sequence with a length of 1, is *not* an atom! An atom doesn't have a length. An atom, and a sequence with a length of 1, are not the same.

2. Everything written within double quotes is a string, and thus a sequence.

3. We've said that the simple form of a sequence is a numbered series of atoms — but less simply, a sequence is a numbered series of elements which can be atoms, or themselves sequences. A sequence can consist of atoms and sequences, of which each element again can be an atom or a sequence, of which … we'll come back to this later.

In our example, we have replaced the 7th element of s, which had been the atom w or 119, with a *sequence* with the length of 1. The only element of that sequence is the atom W or 87, but that doesn't change the fact that it's still a *sequence*.

When we had declared our variable s as sequence, this allowed one of its elements to be a sequence — no problem in line 3, but while s still was a valid sequence, it wasn't a string anymore, and we then ran into a problem with the *puts* stamenent, which expects a string. (When you consider that in a real program there might be hundreds of lines of code between the one that causes the type mismatch problem and the one at which it manifests itself, this shows that declaring a variable as *string* can help with debugging, when this variable is definitely meant to be a string.)

So, in our example, s[7] = 87 works because 87 is an atom (and it also needs to be an integer in the range of 0 to 255), s[7] = "W" does not work because "W" is a sequence which cannot be part of a string — but how do we change s[7] to W without having to look up its code in an ASCII table? It's simple: by using single quotes instead of double quotes:

```
    s[7] = 'W'
```

Single quotes define an atom. `'W'` is the same as 87, `' '` is the same as 32 (the ASCII code for space), etc. `'abc'` is not valid, nor is `''` (two single quotes with nothing between them), because an atom can neither have three parts nor can it be nothing — a sequence can have, or be, either.

Double quotes define a string. "W" is a string with the length 1, "abc" a string with the length 3, and "" a string with the length zero. Or you could call it a sequence instead of string. Again: a string is a sequence, a sequence is a string when all its elements are atoms with integer values of 0 to 255.

Let's look at the two basic operations with sequences, that can also be applied to strings: concatenation, and slicing.

& joins (concatenates) two sequences — the elements of the second sequence are added, one by one, to the first sequence. The result is a sequence whose length is the sum of the lengths of its parts. If the sequences that you concatenate are strings, the result is again a string.

```
    sequence s, h, w
    h = "Hello"
    w = "World"
    s = h & w
```

Oops — puts(1, s) now shows us `HelloWorld`, because we have forgotten to insert a space between the two words. Let's fix this:

```
    s = s[1..5] & " " & s[6..$]
```

In square brackets you can not only write an index, you can write an index range, indicated by two dots, defining not a single element of a sequence, but a *slice*. The $ sign stands for the last index of the sequence.

And yes, even it it may be a bit confusing, with the & operator we *can* add the sequence `" "` to a string, the same way we could have added any other string (`" "` is a string with the length of 1). The atom `' '` or its ASCII code `32` would have worked as well — we can add an atom to a string. What we can *not* do, is *replacing* an atom in a string with a sequence.

Sequences

We’ve said it before, the *sequence* data type is what makes Phix (and Euphoria) so special. And we’ve already said it before, but we have to say it again because it’s really important to be fully aware of it: A sequence is a series of elements which can be atoms and sequences, and each element that is a sequence can again consist of atoms and sequences — *and so on* … there is no limit to the depth, diversity or complexity of a sequence’s structure. You may never have a need for all that complexity, but it is good to know that whatever you can think of, a sequence can handle it.

Each sequence has a length — the number of its elements, independent of what those elements are, or of how many elements they themselves consist.

Before we can take a closer look at this, we have to learn a Phix command that is a bit clumsy, but we have to live with it: *append*. This command add exactly one element to a sequence, whatever the nature of that element is. (In the Phix documentation, also look at the commands *prepend* and *insert*.)

```
sequence s
s = ""
s = append(s, "Hello world!")
s = append(s, "It's me.")
s = append(s, "How are you?")
for j = 1 to length(s) do
  puts(1, s[j] & "\n")
end for
```

What happens here is that we start with an empty sequence, and then we add three elements to it — the length of our sequence is now 3. The loop writes these three elements to the screen — s[1] is the sequence "Hello world", s[2] is the sequence "It's me." etc.

Each of these elements being a sequence, we can address their individual elements, too, by adding a second index after the first one, again in square brackets:

```
s[1][7] = 'W'
```

assigns the atom 'W' (or 87) to the seventh element of the first element of the sequence s.

Had we written this:

```
sequence s
s = "Hello world!"
s = append(s, "It's me.")
s = append(s, "How are you?")
```

then the result would have been something different, and probably not what you had wanted: the line `s = "Hello world!"` creates a sequence with the length of 12, then the following two elements get added, and the resulting sequence has the length 14 — its first 12 elements being atoms, the final two elements being sequences. It would be a valid sequence, though.

As said above, when your sequence contains a sequence that contains … etc., you can address an elements of any desired depth within the sequence’s structure by listing its indices in square brackets

— this element can be sequences, or an atom. And, and the end of that list, instead of a single index may stand a range of indices, like s[3][2][6..8] — addressing the 6th to 8th element of the 2nd element of the 3rd element of sequence s – and this slice of 6th to 8th element can again consist of atoms, or of sequences. (We could also discuss the Phix commands *columnize* and *vslice* here, but we won't — you can look them up in the Phix documentation.)

To repeat: indices are consecutive integers, starting with 1. They can, of course, be variables, either of the type integer, or of the type atom, but with integer values.

You *can* use negative indices, though: they count backwards, from the end of the sequence.

```
    s = "Hello World!"
```

s[-6] is 'W', the same as s[7], and s[-1] is the same as s[$] (you remember, $ as an index stands for the last element). If you use a range of negative indices, you have to put the larger number first:

```
    puts (1, s[-6..-2])
```

writes "World" — or you could have written s[-6..11], or s[7..-2]

But now, back to appending: it is important to understand the difference between the concatenation operator &, which we have met when discussing strings, and the append command.

`s = append(s1, s2)` adds s2, whatever it is, as a single element to the sequence s1 (as do the commands prepend and insert).

`s = s1 & s2` adds all elements of s2 to s1 — using it with strings is only a special case. The length of the resulting sequence is always the sum of the lengths of s1 and s2, as in this example:

```
    sequence s, t
    s = ""
    s = append(s, "Hello World!")
    s = append(s, "It's me.")
    s = append(s, "How are you?")
    t = ""
    t = append(t, "Thank's, fine.")
    t = append(t, "And you?")
    s = s & t
    for j = 1 to length(s) do
      puts (1, s[j] & "\n")
    end for
```

Since an an element of a sequence can be anything, it can even be nothing — that is, it can be an empty sequence.

```
    s = ""
    s = append(s, "Hello world!")
    s = append(s, "It's me.")
    s = append(s, "How are you?")
    s[2] = ""
```

The sequence still has three elements, the second one now being a sequence with the length of zero.

If, on the other hand, you make a *range* of elements empty — you remember, `s[i..j]` — then you *delete* these elements from the sequence. And, this even works when i and j are the same, meaning you can delete a single element this way:

```
sequence s
s = ""
s = append(s, "Hello world!")
s = append(s, "It's me.")
s = append(s, "How are you?")
s[2..2] = ""
```

By this we have deleted s[2], and the remaining sequence now has the length of 2.

One more thing to be aware of with sequences: ultimately every sequence, no matter how it is built, consists of atoms. While each element of a sequence is either an atom or a sequence, if you follow nested sequences to their end, you necessarily arrive at atoms. (Or at empty sequences — these ”nothings” are not atoms, but can still occupy their places in a sequence.)

Here is a little tool that counts the atoms of which a sequence consists of — a nice example of how a function (or a procedure) can call itself recursively. The function atom(x) returns true (1) if the variable is an atom, and false (zero) if it isn’t an atom, that is, if it is a sequence. (Btw, if you do not feel comfortable with recursive function calls, that’s ok, you’ll probably never need them. But if you want to use them, Phix is there for you.)

```
function count_atoms(sequence x)
  integer n
  n = 0
  for j = 1 to length(x) do
    if atom(x[j]) then
      n += 1
    else
      n += count_atoms(x[j])
    end if
  end for
  return n
end function

sequence s
integer n_atoms
s = ""
s = append(s, "Hello world!")
s = append(s, "It's me.")
s = append(s, "How are you?")
n_atoms = count_atoms(s)
print(1, n_atoms)
```

This works with any sequence, walking through all its ramifications independent of their complexity and possible levels of nested sequences. Elements that are empty sequences are not counted. If you do want to include them in the count, counting them the same as atoms, you have to add two lines (the 3rd and 4th one, here) — when it's not an atom then, necessarily, it is a sequence, and when that has the length of zero, include it in the count:

```
    if atom(x[j]) then
      n += 1
    elsif length(x[j]) = 0 then
      n += 1
    else
      n += count_atoms(x[j])
    end if
```

Or you could add a parameter to the function call that lets you decide whether you want to include empty sequences in the atoms count or not:

```
function count_atoms(sequence x, integer incl_empty)
  -- incl_empty = 0 do not count empty sequences, 1 = do count them
  -- incl_empty has to be 0 or 1 - if not 0, set it to 1:
  if incl_empty then incl_empty = 1 end if
  integer n
  n = 0
  for j = 1 to length(x) do
    if atom(x[j]) then
      n += 1
    elsif length(x[j]) = 0 then
      n += incl_empty
    else
      n += count_atoms(x[j], incl_empty)
    end if
  end for
  return n
end function
```

Note the `elsif` — that's not a typo, it really is *elsif*, not *elseif*. Also note in line 4, that in logical expressions the value of 0 stands for false, while any value other than 0 stands for true.

There is a lot more to know about sequences — for instance, that instead of *if s1 = s2* you have to write *if equal(s1, s2)* — but you'll have to find out all about it in the Phix documentation. But, let's stop here — this is not meant to be a Phix programming tutorial, just an introduction to some of Phix's features.

One more note, though: A function returns a single result — but this result does not have to be a single value (that is, an atom), it can as well be a sequence, and that way a function can return any number of results.

To illustrate this we can modify our previous example so that it returns the number of atoms, the number of sequences, and the number of empty sequences within one sequence. I've said let's stop here … but, if you really want to see it, I've added that function as an appendix.

One last remark: We've said that with function and procedure calls, the number of arguments must match — that is, when you declare a subroutine, the declaration must list the arguments with which it will be called. But when an argument is a sequence, this single argument can contain any number of values, de facto allowing you to write subroutines with variable numbers of arguments.

Curly Brackets

Double quotes let you write strings, single quotes let you write characters or their ASCII codes — curly brackets let you write sequences.

Everything within curly brackets is a sequence. Its elements, separated by commas, can be atoms (numbers, or single characters in single quotes), strings in double quotes, or they can be sequences, again in curly brackets — there is no limit to the depth to which curly brackets can be nested.

```
sequence s
s = {}                     -- empty sequence, exactly the same as s = ""
s = {' '}                  -- sequence with length 1, containing a space character
s = {32}                   -- the same as s = {' '}
s = {0,0,0}                -- sequence with length 3, each element has the value zero
s = {"Hello","World!"}     -- sequence with length 2, each element is a string
s = {{72,101,108,108,111},{87,111,114,108,100,33}} -- same as above
```

and so on. Inside the curly brackets you can also use variables:

```
sequence s
s = "Hello"
s = {s,"World!"}           -- same as s = {"Hello","World!"}
```

or arithmetic expressions … there are a lot of possibilities. But even if you do not use curly brackets in your code (they can be quite useful), you'll probably encounter them in error messages, as they are Phix's way of showing you the content of sequences.

Objects

We've seen four data types so far — integers, atoms, strings, and sequences. With integers being a special case of atoms, and strings being a special case of sequences, you wouldn't really need the *integer* and *string* data types, but they are useful to clarify the code and make it easier to debug.

Phix's fifth data type is *object*, and objects, in principle, make the four other data types redundant, because an object can be *anything* — declare a variable as object, and it can be an integer, an atom that isn't an integer, a string, a sequence that isn't a string … and, that's everything a variable can possibly be in Phix.

```
object x
x = "Hello World!"        --  now x is a string
x = length(x)             --  now x is an integer
```

and so on. Had x been declared as a sequence, this last statement would have caused an error.

As with integer vs. atom, and string vs. sequence, the reason for not declaring all variables as objects is to help avoid mistakes, make the code better understandable, and help with debugging. (I do not now about the internal workings of Phix, so I do not know if declaring variables as integers or atoms will result in faster code than declaring everything as objects.)

Talking about debugging: there is a quick way of displaying the content of variables, about which the documentation says: “It is typically used for quick debugging statements that will be removed before the final product is shipped.” I haven’t mentioned it so far, but here it is: the question mark, as an abbreviation for the *print(1, …)* command. Unlike print, ? adds a line break.

```
object x
x = "Hello World!"
? x
x = append("", x)
? x
x = append(x, "It's me.")
? x
x = length(x)
? x
```

(Note: The third line that gets displayed is {"Hello World!",`It's me.`} — the ' apostrophe in *It's me* triggers the string to be enclosed in backticks (ASCII 96), instead of " double quotation marks. This is a slight inconsistency — backticks, as opposed to double quotes, disable escape characters in strings.)

The *puts*, *printf*, *sprint* and *sprintf* commands give you a much better control over how words and numbers are displayed on the screen or written to files, but at the developing and debugging stage, ? is very helpful. Not only is it quickly written, its main advantage is that you do not need to adapt it to the type of the variable you want to display — it works with all kinds of objects, whether they are atoms, strings or sequences, including nested sequences, and gives you information about the nature of your variable.

For instance, look at this — the two strings s1 and s2 are identical:

```
sequence s1, s2
s1 = "Hello"
s2 = {72,101,108,108,111}
? s1
? s2
puts(1, s1 & "\n")
puts(1, s2 & "\n")
```

You will get:

```
"Hello"
{72'H',101'e',108'l',108'l',111'o'}
Hello
Hello
```

How Phix manages to let ? treat s1 and s2 differently I do not know, but when you use ? Phix tries to show the content of the variable in the most appropriate form, from the programmer's point of view. It may be a matter of taste, but the same program in Euphoria would give you

```
{72,101,108,108,111}
{72,101,108,108,111}
Hello
Hello
```

* * *

A long time ago, its developers called Euphoria "A programming language that's powerful, easy to learn, and a lot more fun than other languages," saying it was "Simpler than Basic, More Powerful than C++". Today, few people will remember Basic anymore, and whether the current versions of Euphoria and Phix are more powerful than C or C++ I am not able to say, but the basic principle still holds: that the language is easy to read and to write, and that its *object* data type allows a flexibility that other languages can not offer — indeed, once you know how to get around the stumbling blocks (and we've already heard all about them), it *is* fun to work with those variables that you can turn into any shapes and use for any purposes you want. And yes, Phix also knows associative arrays (called dictionaries), and in those associative arrays not only the values, but also the keys can be *objects* — atoms, strings, sequences …

Everyone needs different things from a programming language, Phix may or may not be what you are looking for — but, it is quickly installed, and you know now how to begin — just give it a try!

Appendix

Here is the promised function that counts sequences, empty sequences and atoms within a sequence, by returning a sequence that contains the three numbers. This is not entirely trivial, and if you have little experience with programming, do not worry if some things do not seem clear at first sight.
I do hope this works correctly!?

```
function count_elements(sequence x)
  sequence n
  n = {0, 0, 0}
  -- n[1] atoms, n[2] empty sequences, n[3] sequences

  for j = 1 to length(x) do
    if atom(x[j]) then
      n[1] += 1
    elsif length(x[j]) = 0 then
      n[2] += 1
    else
      sequence c
      c = count_elements(x[j])
      n[1] += c[1]
      n[2] += c[2]
      n[3] += 1 + c[3]
    end if
  end for
  return n
end function

procedure disp(string text)
  puts(1, text & "\n")
end procedure

sequence s
sequence elements
...
...
elements = count_elements(s)
disp("The sequence contains:")
disp(sprint(elements[3]) & " sequence(s)")
disp(sprint(elements[2]) & " empty sequence(s)")
disp(sprint(elements[1]) & " atom(s)")
```