Neon Programming Language

Documentation

v1.1

This documentation applies to version 4.1 of Neon.

Introduction

Neon is a general-purpose scripting language that is natively concurrent, designed among other things to make concurrent programming possible on machines that do not have a multitasking operating system. The Neon language allows the description of sequential and concurrent computation programs; the Neon interpreter allows them to be executed.

Neon is designed to be easy to understand and easy to use. This documentation is not intended to be pedagogical, but serves as an exhaustive reference for all of Neon's features. This document is also not meant to be read in order, but to organize information into categories.

By "Neon", we refer both to the language in its specifics and to its one and only interpreter.

Programming Environment

The Neon interpreter has two modes: a console mode, which allows entering code and expressions, and an execution mode for running files directly.

Direct Execution Mode

The officially supported file extension for Neon programs is `.ne`. It is recommended to name all Neon programs with this extension. To launch a program in execution mode, simply pass the filename as an argument to the Neon interpreter. The filename may be followed by arguments to be passed to the Neon program.

On the `TI_EZ80` platform, Neon files are AppVars containing the Neon source code as plain text. The names of these AppVars have no extensions. To launch a file in execution mode, put the file's name in the `Ans` or `Rép` variable. To do this, write the filename in quotes on the calculator's home screen and press ENTER. When launched, the NEON program will detect the filename in this variable and execute it.

To facilitate development in Neon, the interpreter is compatible with Python AppVars.

For the Python application to distinguish Python AppVars from regular AppVars, they always begin with the 4 letters `PYCD` followed by the byte `00` (written as `PYCD\x00`). When an AppVar starting with `PYCD\x00` is launched with the Neon interpreter, the latter will simply ignore the first 5 bytes.

In order to specifically identify Neon programs as distinct from Python programs or plain AppVars, it is also possible to start a Neon AppVar with the 4 letters `NEON` followed by the byte `00`.

Exactly as with `PYCD\x00`, the interpreter launches the file ignoring the first five bytes corresponding to `NEON\x00`.

This Neon-specific header could be used to implement a Neon IDE that automatically encodes `NEON\x00` at the beginning of files, exactly as the Python application does with `PYCD\x00`.

Launcher Mode
This mode makes it possible to implement shells and graphical launcher interfaces in Neon for launching Neon programs on the `TI_EZ80` platform. This feature is present on all platforms, but is most useful on the `TI_EZ80` platform.

On the `TI_EZ80` platform:
If the calculator contains an AppVar named `LAUNCHER` beginning with the characters `#NEON` or `NEON\x00`, the interpreter will automatically launch this program as a Neon program.

On other platforms:
If the current directory contains a file named `__launcher__.ne`, it will be automatically executed when the interpreter starts.

Note that if `Ans` (or `Rép`) contains a filename, Neon will always prefer to launch that file directly rather than the launcher.

Console Mode
When the console is ready to receive an expression to evaluate, the cursor is on a new line starting with two chevrons `>>`. If an expression is entered, it will be evaluated, and its result along with its type will be displayed on a new line.

To enter blocks of code, keep in mind that pressing ENTER will send the typed text to the interpreter. You must therefore either use the newline delimiter `;`, or ensure that the line begins with `..` indicating that pressing ENTER will not cause the text to be sent to the interpreter.

On the `TI_EZ80` platform, to enter console mode, the variable `Ans` or `Rép` must contain an empty string, or any other non-string data type.

Table of Contents

Preamble on Syntax and Semantics

When writing a Neon program, you write text — a sequence of characters. The program can be considered simply as this sequence of characters; in that case we talk about syntax.

On the other hand, if we are interested in how these sequences of characters are interpreted by Neon, we talk about semantics.

It is therefore always important to distinguish these two perspectives. Sometimes we will speak of expressions; these are the sequences of characters written in a program. An expression evaluates to an object during execution, and one can associate an expression with the object it evaluates to. For example, if you write the variable `a` in a program, it will evaluate to its own value. Similarly for the expression `123.456`, which evaluates to the float 123.456.

In what follows, we will speak of variables, function names, container field names, etc. These names are referred to by the term identifier. Identifiers must follow precise rules to be correctly read by Neon.

An identifier must necessarily begin with either a lowercase letter, an uppercase letter, or an _. For subsequent characters, digits, an apostrophe ' and a tilde ~ are also allowed.

However, it is recommended to only use the character ~ for module objects.

Finally, there is a list of forbidden variable names as they are already used by Neon's keywords.

Here is the exhaustive list:

```
or
do
if
in
EE
ei
es
tr
xor
and
not
try
for
end
del
NaN
then
elif
else
pass
True
```

```
None
expt
atmc
break
while
False
local
await
import
atomic
except
method
return
foreach
continue
function
Infinity
parallel
```

Neon's syntax is case-sensitive and whitespace-insensitive (spaces and tabs are ignored). Newlines are allowed inside expressions after opening parentheses, opening brackets, and commas.

Part 1: Objects and Variables

All objects usable in Neon programs are grouped within a single C structure called `NeObj`.

Here is the list of all existing Neon object types:

```
Integer
Real
String
Bool
List
NoneType
Exception
Built-in function
Function
Method
Container
Promise
```

1.1 - Variables, Lists and Containers

1.1.1 - Storing Individual Objects with Variables

A variable is a memory slot managed by Neon that can hold an object. Most objects created in Neon have no persistent existence. They are created during the evaluation of an expression, used as an argument to a function or operator, and then deleted.

To retain an object over time, it can be stored inside a variable. A variable is designated by a name. Its name must follow the identifier rules. Many operators allow managing variables, modifying their contents, deleting them, etc.

If a variable does not exist and an expression tries to access its value, an error will be triggered. To create a variable, simply assign a value to it as if it already existed. Example: `myNewVariable = 12`. This alone creates a variable with the value 12.

In addition to variables, there are other ways to retain values over time. There are objects that allow storing a certain number of other objects, accessing them, and modifying them: lists and containers. Their use cases differ.

1.1.2 - A Large-Scale Storage Object: Lists

Lists are objects that have the ability to store an (almost) unlimited number of other objects in an ordered manner. They are used to store large amounts of information, often of the same type. A list is like a sequence of variables, where each element is identified by an index. Each element of a list (thus identified by a certain index) is a memory slot just like a variable. Therefore, the operators that allow interacting with variables also allow interacting with list cells in the same way.

1.1.3 - Grouping Information with Containers

Containers are also objects that contain other objects. Each field of a container is also a memory slot just like a variable, and can be modified using the same operators as those that modify the contents of list cells and variables. Containers are used to group a number of different pieces of information about the same object.

Although these three entities (variables, lists, and containers) have things in common, it is important to understand the fundamental difference between variables and lists/containers.

A variable is indeed a memory slot that can store an object, but unlike lists and containers, a variable **is not** an object. It contains an object. The only way to access a variable is by its name. Thus a variable can contain a list or a container, or any other object, but cannot contain another variable.

A list or a container, on the other hand, is a simple object. It has no name per se, and can be created as the result of an expression or simply deleted when it is no longer needed. To retain it, it must also be stored (in a list, a container, or a variable).

1.2 - Objects

1.2.1 - Memory Organization, Reference Counting, and Garbage Collector

All objects are structures passed on the stack of size one pointer + one byte. Some objects such as numbers, booleans, the None constant, exceptions, and promises are stored solely in this structure. For other objects, the structure contains a pointer to a heap-allocated zone that actually contains the object. When these objects are copied, only the stack structure is copied; the heap block is shared among all copies. Each of these blocks contains a counter that counts the number of references to the block, and the block is freed when this counter reaches zero.

Some objects (containers and lists) contain other objects. This situation can create cyclic objects, making reference counting insufficient.

For this reason, Neon also has a Garbage Collector. The Garbage Collector maintains a list of all containers and lists created. At specific points in the interpreter, a collection is triggered, and Neon marks all objects reachable from still-live variables. Objects not marked during this phase are permanently deleted.

1.2.2 - The `Integer` Type

`Integer` objects are whole numbers. Their size depends on the target system: they are 64-bit integers on 64-bit operating systems, and 24-bit integers on ez80.

`Integer` values can be created as constants in the program source code, combined with operators to generate new ones, or passed to built-in functions to create new ones. See the section on operators and built-in functions for more information.

It is possible to define `Integer` constants in decimal notation, but also in hexadecimal and binary. To define an `Integer` from its hexadecimal representation, precede the hexadecimal notation with the prefix `0x`, and with the prefix `0b` for binary.

For hexadecimal, letters can be either lowercase or uppercase.

1.2.3 - The `Real` Type

`Real` objects are floating-point numbers. As with integers, their size depends on the target system. They are double-precision floats when the OS allows it, otherwise they are 32-bit floats (including on ez80).

`Real` values can be created as constants in the program source code, combined with operators to generate new ones, or passed to built-in functions to create new ones. See the section on operators and built-in functions for more information.

Note: Arithmetic operators on numbers do not all preserve type. While multiplication, subtraction, and addition of two integers remain integers, the division of two integers will always return a decimal number. This behavior of operators is explained in more detail in the section on operators.

1.2.3 - The `Bool` Type

`Bool` objects can only be `True` or `False`. These are the only two possible values. These values can be obtained by writing them literally in a program, or through boolean operators, which will be described in the Operators section.

Any boolean expression (result of comparison operators, logical operators) evaluates to `True` or `False`.

Booleans allow representing truth values, and thus driving conditional branches, loops, etc.

1.2.4 - The `List` Type

A list is an object that stores a finite sequence of arbitrary objects. The precise computer science term for this object would be array. The Neon programming language makes no distinction between these two terms. When referring to a linked list, one explicitly says “linked list.” Lists can be indexed using the following syntax: `list[i]`. The indexing of list elements starts at zero rather than at 1. Thus, `list[i]` does not correspond to the i-th element but to the (i+1)-th. This may seem confusing to someone unfamiliar with it, but it is in fact consistent with the rest of the language. This choice was made to stay in line with the conventions of the vast majority of programming languages. Historically, this convention dates back to the 1970s with the C language in particular, where array indexing expresses an **offset from the first element** rather than access to element number i.

A list can contain any type of object, including itself.

When a list is copied, its elements are not copied. Thus any modification made to a copy of a list will appear in all other copies and the original. To truly copy a list and make it independent of the original, use the `copy` function for a deep copy.

Lists can be obtained in several ways: by receiving the return value of certain functions or operators, or by writing list literals. These are expressions that evaluate to a list.

The syntax for creating a list literal is `[element1, element2, ..., elementn]`. The elements can be any expressions. The list will contain the objects resulting from the evaluation of those expressions (see the section on expressions for more details).

1.2.5 - The `String` Type

Strings are null-terminated ASCII character sequences. They can be created as constants in the program source code, or combined/obtained through other functions. To create a string as a constant, surround the text with double quotes `"` or single quotes `'`. Example:

```
word = 'car'
```
or
```
word = "car"
```

The advantage of these two different syntaxes is that to define a string containing the `"` character, you can use `'` delimiters, and vice versa.

Certain special characters can be defined in strings using the `\` character. Here is the complete list:
`\a` → bell character
`\b` → backspace
`\f` → form feed
`\r` → carriage return
`\v` → vertical tab
`\t` → horizontal tab
`\n` → newline

The length of a string can be obtained with the `len` function, and character number i can be accessed using the syntax `string[i]`. This syntax is the same as for indexing lists.

1.2.6 - The `NoneType` Type

The `NoneType` type designates one and only one object: the constant `None`. It is the only object of type `NoneType`. This constant can be obtained by writing None in a program, or received as the return value of certain functions and operators. This value is a default value often used to indicate that something contains nothing, serves no purpose, or is empty.

1.2.7 - The `Exception` Type

Exceptions are objects that denote types of errors. When an error occurs in Neon, it corresponds to a certain exception. These exceptions can be used to catch certain types of errors via `try .. except`, and to execute certain code depending on the exception raised. There are built-in exceptions, and exceptions that can be created with the `createException` function.

Here is the list of built-in exceptions:
`SyntaxError`: Triggered by a syntax error
`FileSystemError`: Triggered when a file read/write function fails
`UnmeasurableObject`: Triggered by the `len` function on a non-measurable object
`UndefinedVariable`: When trying to use an undefined object
`IncorrectFunctionCall`: When a function call fails
`MemoryError`: Error during resource allocation. This type of error is generally serious
`NonIndexableObject`: Attempt to index a non-indexable object. Only lists and strings are indexable
`IncorrectIndex`: Triggered when the index of a string or list is not a positive integer
`OutOfRange`: When an index exceeds the size of an object
`IncorrectType`: A type is not compatible with a certain operation
`DivisionByZero`: Euclidean division by zero
`UnknownError`: Triggered when there is not enough information about the error
`AssertionFailed`: Triggered when the `assert` function fails
`DefinitionError`: Triggered mainly when a container definition is incorrect with respect to the information the interpreter has about that container type

`KeyboardInterrupt`: Triggered by Ctrl-C in the terminal. On the `TI_EZ80` platform, this error is triggered by pressing the `ON` key
`NotImplemented`: Triggered when calling an unimplemented feature. This can occur when calling functions implemented only for certain platforms, such as `initGraphics` (see the graphics section).

1.2.8 - The `Built-in function` Type

This type groups all functions present in memory when the interpreter loads.

Functions are objects like any others; they can move from variable to variable, be stored in lists, etc. Any object containing a function can be executed as a function.

To execute a function, simply follow the expression containing the function with `(argument1, argument2, argument3...)` with all the arguments you want to send to the function.

Example:
If the variable `myFunction` contains a function and you want to execute this function on arguments `a` and `b`, simply write the expression: `myFunction(a, b)`. This expression will evaluate to the result of the function applied to the objects denoted by `a` and `b`.

In addition to this syntax for calling functions, there is another syntax, object-oriented. Instead of writing `myFunction(arg1, arg2, arg3...)`, you can write `arg1.myFunction(arg2, arg3...)`. These two syntaxes are equivalent.

All functions in the general sense share the same call syntax. Thus, the syntaxes presented here for `Built-in function` type functions also work for user-defined functions and methods.

All information about a built-in function can be obtained by typing `help(function)` in the terminal.

In general, when a function returns nothing, it actually returns None.

Here in alphabetical order is the list of all built-in functions:

`abs(Real)` ⟶ `Real`:
Returns the absolute value of a number.

`append(List, Any)` ⟶ `None`:
This function takes a list and any object as arguments, and appends the object to the end of the list.

`assert(Bool)` ⟶ `None`:
This function takes a boolean as an argument, and raises the AssertionFailed exception if the boolean is False.

`bin(Integer)` ⟶ `String`:
This function takes an integer as an argument and returns a string corresponding to its binary representation. The string is not preceded by the `'0b'` prefix as is the case in Python.

`ceil(Real)` ⟶ `Real`:
Returns the ceiling (round up) of a number.

`chr(String)` ⟶ `Integer`:
This function takes an ASCII code (integer) as an argument and returns the corresponding character.

`clear()` ⟶ `None`:
This function clears the terminal.

`copy(Any)` ⟶ `Any`:
This function returns a deep copy of an object, preserving pointer dependencies within the object.

`cos(Real)` ⟶ `Real`:
Computes the cosine of an angle in radians.

`count(List | String)` ⟶ `Integer`:
This function counts the number of occurrences of a substring in a string, or counts the number of objects present in a list. Specify the list or string first, then the sub-object or substring to count.

`createException(String)` ⟶ `Exception`:
This function is used to create exceptions. It takes a string as an argument and creates an exception with the given name. A new keyword is created with this exception's name, and it is directly accessible using that keyword. The `createException` function also returns the created exception.

`deg(Real)` ⟶ `Real`:
Converts an angle in radians to degrees.

`detectFiles(String)` ⟶ `List`:
This function returns a list of all filenames in the current directory that begin with a given string. It takes the string to compare against the beginning of each filename as its argument.

`eval(String)` ⟶ `Any`:
This function takes a string corresponding to a Neon expression as an argument, and returns the result of evaluating the expression.

`exit()` ⟶ `None`:
This function takes no arguments and exits the interpreter. It returns None.

`exp(Real)` ⟶ `Real`:
Computes the exponential of a number.

`failwith(String)` ⟶ `None`:
This function takes a string as an argument and exits the interpreter while displaying that string.

`floor(Real)` ⟶ `Real`:
Returns the floor (round down) of a number.

`gc()` ⟶ `None`:
This function calls the Garbage Collector.

`help(Any)` ⟶ `None`:
This function displays help related to certain objects or types of objects. Here are the possible arguments to the `help` function:
`help("modules")` → displays all module names present in memory
`help("variables")` → displays all defined variables present in memory along with their type
`help("MyModule")` → displays all objects linked to module MyModule present in memory
`help(my_object)` → displays my_object and its type

For a built-in function, the `help` function also displays the expected argument types, the return type, and a help string explaining how to use the function. For a user-defined function (or user-defined method), it displays the function name, expected arguments, and if a help string has been assigned via `setFunctionDoc`, displays that string.

`hex(Integer)` ⟶ `String`:
This function takes an integer as an argument and returns a string corresponding to its hexadecimal representation. As with the `bin` function, the string is not preceded by the `'0x'` prefix.

`index(List, Any)` ⟶ `Integer`:
This function takes a list and an object from that list as arguments, and returns the index of the first occurrence of the object in the list.

`int(Any)` ⟶ `Integer`:
This function takes an object as an argument and converts it to an integer.

`input(...)` ⟶ `String`:
This function takes a string as an argument, displays it, and waits for text from the user. It returns a string corresponding to the text entered.

`insert(List, Any, Integer)` ⟶ `None`:
This function takes a list, an object, and an index in that list as arguments, and inserts the object at the specified index.

`len(String | List)` ⟶ `Integer`:
This function takes a list or a string and returns its length.

`list(String)` ⟶ `List`:
This function transforms a string into a list whose elements are the characters of the string.

`listComp(String, Integer, Integer, Integer, String, String)` ⟶ `List`:
This function is used to build lists efficiently. It takes as arguments, in order:
- The name of a variable (as a string)
- A start index
- An end index
- A step
- A string corresponding to a boolean expression
- A string corresponding to any Neon expression.

The call `listComp("variable", start, end, step, "condition", "expression")` returns a list built as follows:

```
l = []
for (variable, start, end, step) do
    if (condition) then
        l.append(expression)
    end
end
```

`ln(Real)` ⟶ `Real`:
Computes the natural logarithm of a number.

`loadNamespace(String)` ⟶ `None`:
This function loads into memory a copy of the objects from the module whose name is given as an argument, without the prefix.

`log(Real)` ⟶ `Real`:
Computes the base-10 logarithm of a number.

`log2(Real)` ⟶ `Real`:
Computes the base-2 logarithm of a number.

`nbr(String)` ⟶ `Real | Integer`:
This function takes a string representing a number and converts it to an integer or a decimal.

`ord(String)` ⟶ `Integer`:
This function takes a character as an argument and returns its ASCII code.

`output(...)` ⟶ `None`:
This function takes an unlimited number of objects of any type as arguments, and displays all of them in sequence without a newline.

`print(...)` ⟶ `None`:
This function accepts an indefinite number of parameters and displays the string representation of each parameter, separated by a space. This function also prints a newline. It returns None.

`rad(Real)` ⟶ `Real`:
Converts an angle in degrees to radians.

`raise(Exception, String)` ⟶ `None`:
This function takes an exception and a string as arguments, and exits the interpreter by raising that exception and displaying the string as an error message.

`randint(Integer, Integer)` ⟶ `Integer`:
This function takes two integers: a lower bound and an upper bound, and returns a random integer between these two bounds, with the upper bound excluded.

`readFile(String)` ⟶ `String`:
Takes the name of a text file as an argument and returns its content. On the TI_EZ80 platform, files can only be AppVars.

`remove(List, Integer)` ⟶ `None`:
This function takes a list and an index in that list as arguments, and removes the element at that index.

`replace(String, String, String)` ⟶ `String`:
This function takes three strings as arguments, and replaces all occurrences of the second string in the first string with the third string.

`reverse(String | List)` ⟶ `String | List`:
This function takes a string or a list and returns the reversed object without modifying the original.

`round(Real, Integer)` ⟶ `Real`:
Computes the rounding of a number to the precision given as the second argument.

`safeExec(String, List)` ⟶ `None`:
This function takes a Neon source filename and a list of objects as arguments, and runs the program in an independent environment, passing the list of objects as arguments.

`setAtomicTime(Integer)` ⟶ `None`:
This function changes the process-switching period with the given integer.

`setColor(String)` ⟶ `None`:
This function changes the color of text displayed in the terminal after its call. Available colors are: `"blue"`, `"red"`, `"green"`, `"default"`.

The `default` color is either white in white-on-black mode, or black in black-on-white mode.

On terminals where it is available, red and blue are displayed in bold.

`setFunctionDoc(Function | Method)` ⟶ `None`:
This function takes a user-defined function and a string as arguments, and sets this string as the help message for the function. This help message is displayed when the `help` function is called with this function.

`sin(Real)` ⟶ `Real`:
Computes the sine of an angle in radians.

`sortAsc(List)` ⟶ `None`:
This function sorts a list in ascending order according to lexicographic or numerical order.

`sortDesc(List)` ⟶ `None`:
Same but in the opposite order.

`sqrt(Real)` ⟶ `Real`:
Computes the square root of a number.

`str(Any)` ⟶ `String`:
This function takes any object and returns its textual representation, evaluable by Neon.

`sub(String, Integer, Integer)` ⟶ `String`:
This function expects three arguments: a string and two integers. `sub(string, i1, i2)` returns the substring of string starting at character number i1 up to (but not including) character number i2.

`tan(Real)` ⟶ `Real`:
Computes the tangent of an angle in radians.

`time()` ⟶ `Integer`:
This function returns the number of seconds elapsed since a certain date depending on the system on which the program is running.

`type(Any)` ⟶ `String`:
This function takes an object as an argument and returns its type. Object types are represented as strings. Here are the types returned by the `type` function:
`"Bool"` → boolean
`"String"` → string
`"Integer"` → integer
`"Real"` → decimal number
`"Built-in function"` → built-in function
`"List"` → list
`"Function"` → user-defined function
`"Method"` → user-defined method
`"Exception"` → exception
`"Promise"` → promise
`"unspecified type"` → corresponds to type −1. No object has this type; it is a special value used to describe function signatures. In general, an argument of type `unspecified type` can be of multiple types, which are specified in the function's help string
`"Undefined"` → returned for an undefined object (of `TYPE_EMPTY`)

The type function never returns the type `"Container"` because it directly returns the container's name.

`writeFile(String, String)` ⟶ `None`:
This function takes a filename and a string as arguments, and writes that string to the file with the given name. If the file already exists, its content is replaced. On the TI_EZ80 platform, files can only be AppVars.

1.2.9 - The `Function` Type

In Neon it is possible to create functions that take arguments as input and execute code based on those arguments. When a function is defined, a variable with the function's name is created, with

the corresponding function object as its value. Any function object is callable, just like built-in functions. You can call `help` on it, and add documentation using the `setFunctionDoc` function.

The way to create functions will be detailed in the dedicated section.

1.2.10 - The `Method` Type

A method is also a user-defined object, much like a function. The only difference is that unlike a function where all arguments are local variables to the function and modifying them has no impact on the outside, the first argument of a method can be modified directly by the method. Concretely, once a method has finished executing, the Neon interpreter retrieves the value of the first local variable corresponding to the first argument, and assigns it to the object that was passed as the first argument to the method. Thus any modification made to the first argument inside a method will also apply to the object that was passed as the first argument.

Other than this feature, methods are exactly like functions.

The way to create methods will be detailed in the dedicated section.

1.2.11 - The `Container` Type

Containers are objects that allow neatly storing other objects, each given a name within the container. Each container also has a name that characterizes it among other containers. All containers defined with the same name must have exactly the same fields. For clarity, container names should begin with an uppercase letter. Besides module names, they are the only objects required to start with an uppercase letter.

Example: Defining a Person container

```
myContainer = Person(firstName: "Paul", lastName: "Durand", age: 34, children:
["Peter", "James"])
```

This container is of type `Person`, and contains the fields `firstName`, `lastName`, `age`, and `children`. The first appearance of a container of a certain type in the code fixes the field names for that container type. That is, from the moment the first `Person` object has been defined as above, all containers of type `Person` must contain the fields `firstName`, `lastName`, `age`, and `children`. It is not necessary to define the fields in order; as long as all fields are present, the definition is correct.

As with lists, copying a container does not copy the objects it contains. Thus a container can contain itself, and modifying a field in a container will cause that modification to appear in all copies and the original. To truly copy a container, use the `copy` function.

To access a field of a container, use the `>>` operator. Using the example container defined above, `myContainer>>firstName` refers to the variable containing `"Paul"`.

This syntax can be used both to read field values: `print(myContainer>>firstName)` and to modify fields: `myContainer>>firstName = "Martha"`.

1.2.12 - The `Promise` Type

Promise objects can only be obtained as the return value of launching a process in parallel. Neon allows launching functions in parallel with the `parallel` keyword.

When a function is launched in parallel with `parallel function(arg1, arg2, args...)`, a promise is returned. This promise is an object that serves no purpose while the process is running, other than identifying the process that returned it, since it is unique for each process. As long as the process is running, its type is `"Promise"`. Once the function launched in parallel has finished and returned its value, the promise returned by the `parallel` keyword (and all its copies) stored in variables, lists, or containers will instantly transform into the return value of the function. If the

function returns nothing, it actually returns `None`, and the promise will take the value of `None`. Only user-defined functions can be executed in parallel (not methods, nor built-in functions). Built-in functions are executed atomically, and the `setAtomicTime` function determines the frequency of switching between processes.

The way to manage processes will be detailed in the dedicated section.

Part 2: Expressions

Expressions are at the core of all Neon code. An expression is a syntactic construct that can be evaluated to an object. Every expression evaluates to one of the objects detailed in **Part 1**. An expression describes a computation. Literal constants are the most basic expressions. They are evaluated directly to their associated object. A variable (designated by its name) is also an expression, and is evaluated to the value of the variable.

More complex expressions can be created by combining other expressions using operators and functions.

These operators and functions expect arguments (or operands), which at the time of evaluation (or execution) of the function or operator must be objects. To send an object as an argument to a function or operator, simply write an expression that evaluates to the desired object in place of the argument.

2.1 - Operators

Here is the list of all operators and their effect on objects of different types:

2.1.1 - Binary Operators

+ :
This operator adds two numbers, concatenates two strings, and concatenates two lists.

***** :
This operator performs multiplication between two numbers. The product of an integer `n` and a list returns the concatenation of that list with itself `n` times. The same applies to the product of a string and an integer.
The product of a list (or string) and a boolean behaves as if the boolean were interpreted as `1` or `0`.

- :
This operator performs subtraction between two numbers.
Subtracting an integer `n` from a string removes the last `n` characters of the string.
Subtracting one string from another returns the first string with all occurrences of the second string removed.

/ :
Although it might seem surprising, this operator simply performs division between two numbers, nothing more. It is true that one could imagine uses of the `/` operator for all sorts of types, but one must sometimes be reasonable.

****** :
Exponentiation operator. `a ** b` returns `a` to the power of `b`.

== :
Object comparison operator. Returns True if and only if the two objects are equal.

!= :
Returns True if the two objects are different.

`>=` :
Comparison between two numbers. Returns True if and only if the left operand is greater than or equal to the right operand.

`<=` :
Comparison between two numbers. Returns True if and only if the left operand is less than or equal to the right operand.

`<` :
Comparison between two numbers. Returns True if and only if the left operand is strictly less than the right operand.

`>` :
Comparison between two numbers. Returns True if and only if the left operand is strictly greater than the right operand.

`and` :
Performs a logical AND between two booleans. This operator is lazy: if the left operand evaluates to `False`, the right operand is not evaluated and the operator returns `False`.

`or` :
Performs a logical OR between two booleans. This operator is lazy: if the left operand evaluates to `True`, the right operand is not evaluated and the operator returns `True`.

`xor` :
Performs a logical EXCLUSIVE OR between two booleans.

`=>` :
Returns the result of the logical implication between two booleans.

`=` :
Assignment operator. The left operand must be either a variable, a list index, or a container attribute; the right operand can be any object. This operator changes the value stored in the left operand. An assignment returns None.

If the left operand is an existing variable, its contents are simply modified. Otherwise, if the variable does not yet exist, it is first created.

`->` :
This operator is also an assignment operator. The object is placed on the left, and the variable/list index/container attribute on the right. Unlike the = operator, this operator also returns a copy of the assigned value. This allows chained assignments: `6 -> a -> b -> c`.

`+=` :
Binary operator; `a += b` corresponds exactly to `a = a + b`.

`-=` :
Binary operator; `a -= b` corresponds exactly to `a = a - b`.

`/=` :
Binary operator; `a /= b` corresponds exactly to `a = a / b`.

`*=` :
Binary operator; `a *= b` corresponds exactly to `a = a * b`.

`%` :
This is a binary operator that expects two integers. It returns the remainder of the Euclidean division of the left operand by the right operand.

`//` :
This is a binary operator that expects two integers. It returns the quotient of the Euclidean division of the left operand by the right operand.

Although the division operator is a simple operator between numbers, there is a special behavior here. When one of the operands is an integer `a` and the other operand is a string of size `n`, the operator returns a new string composed of the first `n//a` characters of the original string.

`<-` :
This operator expects a string on the left and any object on the right. The operator creates a variable with the name given on the left, even if the name is not a valid identifier, and assigns the object given on the right to it. If the variable already exists, the operator simply changes its value. The operator also returns a copy of the object.

`EE` :
This operator corresponds to a power-of-ten multiplication. `a EE b` equals `a * 10 ** b`.

`in` :
This operator expects any object on the left and a list on the right, and returns True if and only if the list contains the left operand.

`<->` :
This operator expects variables/list indices/container attributes on both left and right, and swaps their values.

2.1.2 - Unary Operators

`-`: This operator can also be used as a unary operator. The operand is placed on the right. Returns the opposite of a number.

`del` :
This operator takes a variable on the right and deletes it.

`@` :
This operator is a unary operator that expects a string on the right. It returns the value of the variable whose name is the given string.

`&` :
This is a unary operator that expects a variable on the right. The operator returns the name of the variable.

`++` :
Unary operator, expects a variable, list index, or container attribute on the left. `a++` corresponds exactly to `a += 1`.

`--` :
Unary operator, expects a variable, list index, or container attribute on the left. `a--` corresponds exactly to `a -= 1`.

`not` :
Unary operator, operand on the right. Performs a logical negation.

2.1.3 - Special Operators

`parallel` :
This operator is a special operator that does not take an object as input but an expression. The `parallel` operator must receive a user-defined function call on the right, and launches that function call in parallel, in a new thread of execution.

Example: `parallel f(arg1, arg2, arg3)` launches function `f` in parallel on arguments `arg1`, `arg2`, and `arg3`. Further details will be given in the section dedicated to multitasking.

`.`: This is also a special operator that expects an object on the left and a function or method call on the right. Writing `object.function(arg1, arg2)` corresponds exactly to `function(object, arg1, arg2)`.

`>>`: This is also a special operator that expects a container on the left and a container field name on the right.

2.1.4 - Operator Precedence

All operator precedences are distributed across 9 levels from 0 to 8. The highest level corresponds to the least-prioritized operator, which will be evaluated last.

Level 0: `>>`
Level 1: `-` (unary)
Level 2: `&`, `@`, `.`,
Level 3: `**`, `++`, `--`, `EE`
Level 4: `*`, `/`, `%`, `//`
Level 5: `+`, `-`
Level 6: `==`, `!=`, `<=`, `>=`, `<`, `>`, `in`
Level 7: `and`, `or`, `xor`, `not`, `=>`, `parallel`
Level 8: `=`, `+=`, `-=`, `*=`, `/=`, `<-`, `->`, `del`, `<->`

2.2 - Functions

Functions are objects that produce a behavior, often an output, and generally from arguments. If an object `myFunction` is a function (whether a user-defined function or a built-in function), the function can be executed on arguments `a1`, ..., `an` via the following syntax: `myFunction(a1, a2, ..., an)`. When this expression is evaluated, the resulting object is the output produced by the function. Some functions produce no output (or at least that is not their purpose); these functions return `None`.

Functions can of course be composed with each other and with operators.

Part 3: Program Structure

As stated in the introduction, Neon is a language that allows describing sequential computation programs. This means the Neon language can describe the execution of a series of simple tasks. This is also called a program.

A program is an assembly (concatenation and composition) of code blocks. A code block represents a task to be executed. There are many different types of code blocks, each with its own characteristics.

The most basic code block is an expression. Considered as a code block, an expression is not important for the result it returns, but for the actions it produces during evaluation. When an expression is written as a code block, the expression is evaluated, but its final result is ignored. If you write the following program, consisting of a single code block that is an expression:

```
2 + 3
```

the value 5 will indeed be created, but it will be ignored and deleted. Even if this expression is a valid code block, it serves no purpose as a code block because it produces no behavior. The state of the program (variables, memory) is the same after and before its evaluation.

On the other hand, if you write the following program, again consisting of a single code block that is an expression:

```
myVariable = 2 + 3
```

the value created by the expression 2 + 3 will have been used since the variable `myVariable` now contains it. However, the value returned by the entire expression `myVariable = 2 + 3` (`None`) is once again ignored.

A Neon program is a succession and composition of code blocks. To execute two code blocks in sequence, either separate them with a newline or with a semicolon `;`. Example of a program composed of a succession of simple tasks:

```
myVariable = 2 + 3
myVariable *= 7
myVariable --
print(myVariable)
```

From these basic code blocks, more complex ones can be constructed.

3.1 - Conditional Blocks

Conditional blocks are used to execute code under certain conditions. While code blocks made of expressions in sequence always execute, code inside conditional blocks only executes if a specified condition is true.

A complete conditional block is written as follows:

```
if (boolean expression) then
    code to execute
elif (another boolean expression) then
    other code to execute
⋮ arbitrary number of elif blocks
elif (another boolean expression) then
    other code to execute
else
    other code to execute
end
```

Here is how it is interpreted: the boolean expressions are tested one by one in order. First the one from `if`, then the one from the first `elif`, etc. For the first of these expressions that evaluates to `True`, the code just below it is executed and the conditional block is exited.

If none of the conditions evaluates to `True` (i.e., all conditions evaluate to `False`), the code inside the `else` block is executed.

As stated above, this is a complete conditional block.

A valid conditional block must necessarily begin with an `if` block, then may contain any number of `elif` blocks (including zero), and then may contain an `else` block.

The following conditional blocks are valid:

```
if (1+1 == 2) then
    print("true")
end
```

```
if (1+1 == 0) then
    print("true")
else
    print("false")
end
```

3.2 - `while` Loops

Loops are structures that repeat the execution of a certain code block depending on different parameters.
A `while` loop repeats the execution of a code block as long as a certain condition is true. Here is the syntax of a `while` loop:

```
while (boolean expression) do
    code to execute
end
```

It is important to note that the condition is always tested **before** the code block is executed.

Several facts follow from this:
- The code inside the block can always assume the condition is true
- If the condition is false upon entering the loop, the code will never execute

3.3 - `for` Loops

`for` loops allow executing a code block while varying the value of an integer, called the variant, over a certain range.

The complete syntax of a `for` loop is:

```
for (variant, start, end, step) do
    code to execute
end
```

The `start`, `end`, and `step` fields must be expressions evaluating to integers, and the `variant` field must be a variable.

Upon entering the `for` loop, the variable used as the variant becomes local to the loop. That is, the previous value of the variable is no longer accessible, and the variable will take the new value used in the `for` loop. When the `for` loop is finished, the previous value of the variant is restored. If the variable used as the variant was not defined before the `for` loop, it becomes undefined again. Section 4.3 details local and global variables further.

The behavior of the `for` loop is simple: the code to execute inside the block will run for each different value of the variant, starting from `start`, up to (but not including) `end`, incrementing the variable by `step` each iteration.

The variable used as the variant is updated each iteration, which means that even if it is modified inside the code, it will be restored as if nothing happened at the end of the iteration, and the loop will not be affected.

It is not always necessary to specify the step and start value of the loop. There are two other ways to define a `for` loop:

```
for (variant, start, end) do
    code to execute
end
```

In this case, the default step is 1.

```
for (variant, end) do
    code to execute
end
```

In this case, the default step is 1 and the starting value of the loop is 0.

3.4 - `foreach` Loops

`foreach` loops are based on the same principle as `for` loops: they allow executing a code block for each value in a list or each character in a string, in ascending index order.

Here is the syntax:

```
foreach (element, iterable) do
    code to execute
end
```

As with `for` loops, the variable `element` is local to the loop and is restored at each new iteration.

3.5 - Control Statements

Control statements are code blocks in their own right, just like expressions. They must therefore be separated from other code blocks by a newline or a semicolon.

3.5.1 - `break` Statement

The `break` statement must be used exclusively inside loops. When a `break` statement is encountered, execution immediately exits the innermost loop containing the `break`, and continues just after that loop.

3.5.2 - `continue` Statement

The `continue` statement must be used exclusively inside loops. When a `continue` statement is encountered, execution jumps to the beginning of the innermost loop containing the `continue`. In this operation, the loop condition is re-checked, so if the condition was false at the time of `continue`, it will have the same effect as a `break`.

3.5.3 - `pass` Statement

The `pass` statement does absolutely nothing.

3.6 - Error Handling

Neon has an error handling system. When an error occurs, an exception is triggered. It is possible to detect the triggering of exceptions using `try ... except` blocks, and to execute code based on the exception raised.

The exceptions that exist by default are detailed in section 1.2.7.

It is possible to manipulate exceptions as objects and to create new ones using the `createException` function.

It is also possible to voluntarily trigger exceptions using the `raise` function (see how to use it in section 1.2.8).

Here is how to write a `try ... except` block:

```
try
    code to execute
except (Exception1, Exception2, ...) do
    code to execute
```

```
⋮ arbitrary number of except blocks
except (Exception3, ...) do
    code to execute
end
```

When a `try ... except` block is executed, the code inside the `try` is first run. If everything goes well (i.e., no exception is raised), the `try ... except` block finishes when the code inside the `try` is done.

However, if an exception is raised during the execution of the code inside the `try`, execution stops at the point of the exception, and the first `except` block whose listed exceptions match the one raised will be executed in full, normally. When that `except` block finishes, the `try ... except` block also ends.

Note that Exception1, Exception2, ... must be expressions evaluating to objects of type `Exception`. These expressions can for example be constants corresponding to exceptions (DivisionByZero, etc.) or variables containing exceptions.

The number of exceptions that can be specified for an `except` block is unlimited and can even be zero. When no exception is specified between the parentheses, the `except` block is executed regardless of which exception was raised.

Part 4: Defining Functions and Methods

In addition to functions defined by default by the Neon interpreter (of type Built-in function), it is possible to define two other types of functions: user-defined functions (of type Function) and methods (of type Method).

A function is a code block that takes arguments as input, produces output or behavior, and optionally returns an output.

Defining functions is what should make Neon code as readable as possible, close to natural English text.

4.1 - Defining Procedures

The conceptually simplest functions are procedures. These are functions that take no arguments and return no output. Such functions always perform the same action when called. Defining these functions serves only to make code more readable and potentially more concise, by calling functions with clearly defined names rather than executing a block of code directly.

Neon does not distinguish procedures from full functions. A procedure is simply the term for functions without arguments that return nothing. A procedure is defined as follows:

```
function myProcedure() do
    code to execute
end
```

A procedure returns `None`. All functions that do not contain a `return ()` return `None`.

4.2 - Basic Functions

Let us now see how to pass arguments to a function and return a result from a function.

```
function myFunction(arg1, arg2, arg3) do
    code to execute
end
```

This function takes three values as arguments, and these values are stored in variables `arg1`, `arg2`, and `arg3`. The code inside the function can therefore use these variables knowing their values are those of the arguments passed to the function.

Unlike code written anywhere in a program, the code of a function can use an additional code block: the `return ()` block. This code block can either contain an expression: `return (expression)`, or remain empty: `return ()`. The expression can be of any type. When a `return ()` block is encountered inside a function, it ends the function. This means that any code located after a `return ()` is never executed.

When the `return ()` block contains an expression, this expression is evaluated at the moment the `return ()` is encountered, and the obtained value is returned as the function's output.

When the `return ()` block is empty, the returned value is simply `None`.

Placing a `return ()` or a `return (None)` at the very end of a function is exactly equivalent to placing nothing there.

4.3 - Local and Global Variables

In order not to interfere with a program's variables and to make the code executed inside a function invisible to the calling code, there are different levels of variable locality. These locality levels imply that certain variables can have multiple different values at the same time, only one of which is accessible.

In reality, during a Neon program, a variable is like a stack. When you modify a variable or access its value, you always manipulate the value at the top of the stack.

When a variable becomes local to a new code block, a new value is pushed onto the stack, and it is this value that the code block manipulates. At the end of the code block that used the variable as a local variable, the used value is popped, and the previous value becomes the main value again.

Code blocks that have the ability to make variables local are:
- Conditional blocks
- `for`/`foreach` loops
- User-defined functions
- Methods

When a function is called, the variables used as arguments to that function are automatically made local to that function. The previous values of these variables are therefore preserved.

In addition to the variables used as arguments, it is possible to make any variable local using the `local ()` code block. This code block expects variables as arguments, separated by commas. The number of variables is unlimited. `local (var1, var2, var3...)` makes all variables between the parentheses local. In this case, the values of these variables will be restored at the end of the innermost code block in which `local ()` was located.

As mentioned in sections 3.3 and 3.4, the variables used as variants in `for` and `foreach` loops are also automatically made local.

4.4 - Methods

Methods are functions with an additional feature. In a function, if you modify the values of the variables used to receive the arguments, this obviously has no impact on the arguments themselves.

In the case of a method, things are slightly different.

At the end of a method, the value of the variable used to store the first argument is automatically assigned to the object that was passed as the first argument. Thus, any modification made to the first argument inside a method will also be effective outside the method.

Apart from this feature, methods are exactly like functions.

To define a method, simply use the keyword `method` instead of the keyword `function` when defining.

4.5 - Advanced Argument Passing Methods

The classic way to pass arguments to functions is to separate the argument expressions by commas: `function(exp1, exp2, exp3)`. In reality there are much more advanced features.

4.5.1 - Passing Arguments Out of Order

Sometimes certain functions take many arguments as input, and it is difficult to remember the exact order in which to specify them. Moreover, calling such functions can be quite complex to read: consider the hypothetical example of the following function: `function drawFilledRect(x, y, width, height, fg_r, fg_g, fg_b, fg_a, bg_r, bg_g, bg_b, bg_a)`.

To specify arguments out of order, simply indicate the name of the argument being specified, followed by `:=` and then its value.

Note: It is not mandatory to actually give the arguments in a different order; this is just a possibility and the main use case.

When some arguments are given with their name as shown above, it is not necessary to do so for all other arguments. The arguments with specified names are assigned first, and the remaining unnamed values are distributed in order to the remaining arguments.

Example with the following function: `function f(a, b, c)`

All of the following calls to function `f` correspond to the classic call `f(1,2,3)`:
f(a := 1, c := 3, b := 2)
f(c := 3, 1, 2)
f(1, c := 3, 2)

4.5.2 - Optional Arguments

Optional arguments allow defining functions for which it is not necessary to specify all arguments when calling them. When defining an optional argument, a default value is provided at the time of function definition, to be given to that argument in case the function call does not provide a value for it.

When defining a function expecting classic arguments, simply write: `function myFunction(arg1, arg2, arg3)` separating argument names with commas.

To define an optional argument, simply follow the argument name with `:=`, then with the expression that will evaluate to the default value.

Example: `function myFunction(required1, optional1 := expression, required2...)`

4.5.3 - Unlimited Number of Arguments

It is also possible to define functions that accept an unlimited number of arguments. For this, use the special operator `...` in place of an argument name.

Example: `function myFunction(normal arguments, ...)`

This time, the `...` must literally be written as-is and is not a shorthand notation in this document.

When a function can receive an unlimited number of arguments, only the values that could not be assigned to normal arguments are counted as extra arguments. Indeed, arguments specified by name and optional arguments are assigned to their variables first, then unnamed values are assigned in order to the remaining arguments. Only the values that could not be assigned during the preceding phases will be counted among the extra arguments.

When calling a function with an unlimited number of arguments, a special local variable is created. This variable is a list containing all values that could not be assigned to normal arguments (i.e., the values counted in `...`), and is accessible under the name `_local_args_`.

4.5.4 - Truly Optional Arguments

When defining a function that accepts an unlimited number of arguments, it is also possible to define arguments after the `...`. These arguments must be optional, and are called truly optional arguments.

Example: `function f(a, b, ..., c := 5)`

Since `...` encompasses all values passed to the function beyond the normal arguments, the only way to give a value to a truly optional argument is to specify it using the `:=` syntax.

4.6 - Higher-Order Programming, Closures

It is possible to take advantage of how Neon evaluates and defines functions to obtain behavior similar to closures found in most programming languages.

During the definition of a function, closures allow saving the values of variables that are not local to that function but are local to functions inside which our function is defined.

Example:

```
function plus(value) do
    function myFunction(a) do
        return (a + value)
    end
    return (myFunction)
end
```

This function takes a number as an argument and returns a function that adds that first number to another number. At least, that is what we would want this function to do. However, written as is, this function does not work. Indeed, the function uses the local variable `value`, which exists at the time of defining `myFunction`, but is destroyed when the function `plus` returns its value.

Closures were invented to solve this problem. For several reasons, Neon does not have such a system. The first reason is transparency: Neon does not save/restore variable values behind programmers' backs. Also, Neon already has a relatively complex argument-handling system, and there was no need to make it heavier with a closure system. However, the way Neon is coded would allow for such a system to be easily implemented if the need arose.

Nevertheless, it is possible to simulate this behavior.

Thus, a call to the function returned by `plus` would use an undefined variable and trigger an error.

However, it is indeed possible to code this function in Neon, in a slightly different way. To do this, it is important to understand how the interpreter defines functions.

There are two stages to defining a function.

The first stage takes place at the level of the purely syntactic analysis of the function, before the program starts executing. At this stage, Neon records the function's code, detects the arguments it needs, checks whether these arguments are optional, and finally creates a partial function object. This function object is partial because it does not yet contain the default values of the optional arguments, as these values are not yet known at this stage.

The second stage takes place during the execution of the program, at the moment when the function definition block is passed. At this stage, the default values of optional arguments can be evaluated, and are then stored in a complete function object that is assigned to the variable whose name is that of the function.

Thus, the default value of optional arguments is evaluated exactly once at the time of function definition, and stored in the function object. It is therefore possible to have different functions even though they come from the same definition block, because the functions were defined at different times and thus do not have the same default values for their arguments.

Exploiting this characteristic, here is a correct version of the `plus` function mentioned above:

```
function plus (value) do
    function myFunction(a, value := value) do
        return (a + value)
    end
    return (myFunction)
end
```

Here, we use optional arguments to capture the value of the `value` variable at the time of the function's definition. When the function returned by `plus` is called, since the optional argument will not be provided, the variable `value` will automatically receive the value evaluated at the time of the function's definition — the one that needed to be saved.

4.6 - Modular Programming

When writing programs of significant size with distinct, well-defined components, it is common practice to split the program into modules. In Neon, a module is a set of variables and functions sharing the same prefix.

4.6.1 - The ~ Character

To identify all functions and variables belonging to a module, all of their names must share the same module prefix. A module prefix takes the form: `NameStartingWithUppercase~`.

Example: To create three elements `a`, `b`, and `c` belonging to the module `Module`, simply name them `Module~a`, `Module~b`, and `Module~c`.

To create a module, simply create a variable or function whose prefix is the module name.

This prefix-based syntax facilitates module recognition by Neon. Thus, by typing `help("modules")`, the `help` function will list all defined modules, and similarly by typing `help("ModuleName")`, the `help` function will list all objects whose name begins with `ModuleName~`.

4.6.2 - The `loadNamespace` Function

The `loadNamespace` function takes a string corresponding to a module name as an argument, and creates a copy of all objects from that module without the module prefix. Thus, after a call to `loadNamespace`, it is no longer necessary to write module prefixes before the names of the loaded module's elements.

However, it is important to understand that `loadNamespace` only acts on elements already defined at the time it is called.

4.6.3 - Operator Overloading and Display Overloading

Neon provides the ability to overload certain operators as well as display functions on certain container types. It is possible to configure Neon so that using certain operators on certain container types executes a user-defined Neon function rather than the original operator.

The overloadable operators are:

```
+ : add
- : sub
/ : div
* : mul
% : mod
// : eucl
** : pow
- (unary) : minus
in : in
```

To overload an operator on a certain container type `MyContainer`, look up the name the operator carries in the list above, and define the function `MyContainer~name`, where `name` is the operator's name in the list above. This function will define the action performed by the operator on these objects.

In addition to operators, the `str` function can be overloaded by defining `MyContainer~str`, and the display (`print`, `output`, and the console display) can be overloaded by defining `MyContainer~repr`.

If at least one of the arguments of an operator is a container of a type for which that operator is overloaded, the overload function will be called.

4.6.4 - The `import` Keyword

The Neon language provides a code block called `import ()` that allows executing the contents of Neon programs from their names. The `import` keyword only works with files whose extension is `.ne`. To execute a Neon file with extension `.ne`, simply pass the filename without the extension as the argument to `import`.

For the TI_EZ80 version of Neon, since Neon files are AppVars without extensions, the full filename must be written to import it.

`import` can receive an unlimited number of arguments. As a code block in its own right, it must be separated from other code blocks by a newline or a semicolon, just like `return ()`.

Part 5: Concurrent Programming

5.1 - The Process View

In the previous sections, when describing the behavior of a program, we spoke of it as a sort of read head traversing the program and executing the instructions it encounters. This view corresponds fairly well to the actual flow of program interpretation. This read head, implicitly mentioned when describing a program's behavior, is in some sense the personification of program execution.

From now on, with concurrent programming, we will create programs with multiple different read heads — that is, programs simultaneously executing different portions of code. We say that such a program has multiple threads of execution.

Before going further, it is essential to understand how such a thing is possible, and especially how it is implemented in Neon.

Many computers or electronic devices have only a single processor with a single core. However, almost all operating systems need the ability to execute multiple programs at once. These operating systems then manage what is called interleaving between processes. The processor executes a few instructions of one process, then a few instructions of another, then returns to the first, etc. It alternates execution between all processes. This juggling gives the illusion that these processes are executing simultaneously.

Since Neon aims to run on any platform, it makes no assumptions about the operating system. Therefore, it cannot use such interleaving features.

That is why the Neon interpreter manages interleaving between its own processes itself. This characteristic has advantages, but also disadvantages.

The primary advantage is the ability to run multiple programs at once on devices where it would otherwise be impossible. The fact that the interpreter has direct control over interleaving also allows better control over certain parameters such as the time spent executing a process before switching to the next.

However, on devices with multiple cores for simultaneous execution, the Neon interpreter will be unable to distribute tasks across these different cores.

Thus, a Neon program written in a concurrent manner will never be faster than its fully sequential version. The concurrent features of Neon should only be used when they simplify the writing of the program.

5.2 - The `parallel` Operator

The `parallel` operator allows launching a function in parallel, that is, creating a new thread of execution, a new process executing the specified function with the specified arguments.

This operator is a unary operator that expects a user-defined function call on the right.

Example: `parallel function(arg1, arg2, arg3)`

The `parallel` operator applied to a user-defined function call creates a new process, adds this process to the execution queue (meaning its execution will begin when its turn comes), and returns a promise for this process.

5.3 - Process Return Values via Promises

Promises are objects returned by the `parallel` operator and are of type `'Promise'`. This operator is the only way to obtain objects of type `'Promise'`.

When created, all processes are assigned an identifier. This identifier is unique among running processes, but if a process has finished, its identifier can be reassigned to another process later.

Launching a process with the `parallel` operator creates a unique promise associated with that process via its identifier. This promise can be used to identify the process relative to others.

Promises do not only serve to identify processes, but also allow processes to return a result. Indeed, a process is created from a function call. When the process finishes, it can return the return value of the function it was called with. This return is made via the promise retrieved at process launch.

While the process is running, the returned promise is of type `"Promise"`. Once the process has finished, all promises associated with that process (the original promise and those obtained from the

original promise) stored in variables, lists, or containers will instantly transform into the return value of the process.

5.4 - Passive Waiting

When programming with processes, it is common to create programs that wait. To wait, the only way to do so is a loop like this:

```
while (not condition) do
    pass
end
```

This way of waiting is called active waiting, because the program actively runs while waiting. It is often relatively inefficient to wait this way; to waste computation time doing nothing. Indeed, this kind of structure is almost always used with concurrent programming, and the condition can only become true during the execution of another process.

That is why Neon provides a second way to wait, but passively. This means that if the condition allowing the wait to end is not met, execution immediately passes to the next process. The interpreter favors the execution of other processes over the one that is waiting.

Waiting is done using the syntax `await(condition)`. The execution of `await(condition)` triggers a passive wait **until the condition is true**. The condition must be a boolean expression.

`await()` is a code block in the same way as `import()` and `return ()`. It must be separated from other code blocks by a newline or a semicolon.

5.5 - Process-Local Variables

In order not to interfere with each other, processes each have a context: each variable potentially has a different value in each process. When a variable is global, it has the same value in all processes, and a modification of the variable by one process will be visible in all processes.

However, as soon as the code of a process localizes a variable (by using it as a function argument or by calling `local`), the concerned process will manipulate a personal version of the variable. When the variable is no longer local in the process, it will again manipulate the global version of the variable.

Since Neon's processes are actually executed in sequence rather than in parallel, when a variable is local to a process, that process must save/restore it between the moment it gets its turn and the moment it must hand off to another process.

Each process has a list of variables local to it. When resuming execution, it restores its local version of the variable, works with it, then saves its local version back and restores the global version when switching to the next process.

5.6 - Atomic Blocks

The interleaving between processes managed by the interpreter can make the execution of certain code sequences unpredictable. Indeed, in a program, it is common to make assumptions from one line to the next based on already-verified conditions. However, when other processes can be executed at any moment between two lines, this kind of code can easily become incorrect.

Atomic blocks ensure that a certain code sequence executes atomically, meaning it will not be interrupted at any point and will execute in one go, without any other code being able to run at the same time as the code in the atomic block.

An atomic block is written as follows:

```
atomic
    code to execute
end
```

5.7 - Interleaving System Functions

The switching from one process to another by the Neon interpreter does not occur at arbitrary moments. It is performed by a function called `neon_interp_yield`. This function is called in only two places: during a call to Neon's evaluation function and during a call to the execution function of a Neon code block.

Each time `neon_interp_yield` is called, a counter is decremented. As long as this counter does not reach zero, the function does nothing else, but when it reaches zero, `neon_interp_yield` pauses the current process and resumes another one.

5.7.1 - The `setAtomicTime` Function

The number of times `neon_interp_yield` decrements the counter before switching to the next process is stored in the variable `ATOMIC_TIME`. This is the default value of the atomic counter for a process. By default, `ATOMIC_TIME` is 1500, meaning that for each process, after 1500 calls to `neon_interp_yield`, it will switch to the next process.

The `setAtomicTime` function allows modifying this value (1 is the smallest possible value).

Part 6: Additional Features

6.1 - Program Arguments

When calling a program from the command line, it is possible to pass arguments to it separated by spaces. When Neon is used in execution mode (i.e., by passing a filename as the first argument to the interpreter), the subsequent arguments on the command line are captured by Neon and stored in the list `__args__`. This variable is then accessible by the program for processing the arguments.

6.2 - Predefined Variables and Constants

6.2.1 - The `Pi` Constant

Neon has a constant `Pi` with the value `3.141592653589793`. This value is rounded according to the precision offered by floating-point numbers.

Certain environment variables are predefined by the interpreter to allow the running program to obtain information about its environment.

6.2.2 - The `__name__` Variable

This variable allows the program to know the name of the file in which it is written. More precisely, in the main program (code located in the file launched from the command line), the variable `__name__` has the value `"__main__"`. In an imported file, this variable changes its value to the name of that file. When execution returns to the main file, the value of this variable becomes `"__main__"` again.

6.2.3 - The `__platform__` Variable

This variable allows knowing the operating system and architecture for which the Neon interpreter being used was compiled. The different possible values of this variable are `"LINUX_AMD64"`, `"WINDOWS_AMD64"`, and `"TI_EZ80"`.

The value of this variable is visible in the text displayed at the launch of console mode.

6.2.4 - The `__version__` Variable

This variable is a string representing the version of the Neon interpreter being used. The value of this variable is visible in the text displayed at the launch of console mode.

6.3 - The Special Variable `Ans`

In console mode, each time an expression is entered in the terminal, the variable `Ans` takes the value of the result of that expression.

Part 7: Non-Standard Extensions

In addition to the core language containing everything documented so far in this document and available on all platforms, it is possible on certain platforms to import additional functions not available on other platforms.

7.1 - The Graphics Extension for the TI_EZ80 Platform

This extension is somewhat misnamed, since it is not only graphical: it allows both drawing on screen and managing keyboard input.

When the interpreter loads, nothing defined in this extension is accessible; you must first call the function `initGraphics` (with no parameters) to initialize the extension in memory.

On platforms that do not support the graphics extension, calling `initGraphics` will raise the `NotImplemented` exception.

This extension defines a set of functions and container types that allow creating graphical objects and displaying them.

The way in which container type pre-definitions work here is somewhat special, and it is important to understand how it works.

As explained earlier in this documentation, it is normally sufficient to *write* a container of a certain type for that type to be defined. It is never necessary to *explicitly define* a container type.

The type is defined based on the first object of that type encountered by the interpreter.

The problem is that certain graphics functions take graphical objects (circles, rectangles, lines) as arguments which follow a very precise definition, with specific fields, etc. For the graphics functions to efficiently recognize whether an object is a circle, a triangle, etc., all graphical objects are pre-defined when the extension loads.

The container types defined upon a call to `initGraphics` are:

- `Point(x, y)`: `x` and `y` of type `Integer` or `Real`
- `Circle(x, y, radius, color, filled)`: `x`, `y`, and `radius` of type `Integer` or `Real`, `color` of type `Integer`, and `filled` of type `Bool`
- `Rect(x, y, width, height, color, filled)`: `x`, `y`, `width`, and `height` of type `Integer` or `Real`, `color` of type `Integer`, and `filled` of type `Bool`
- `Line(x0, y0, x1, y1, color)`: `x0`, `y0`, `x1`, and `y1` of type `Integer` or `Real`, and `color` of type `Integer`
- `Text(text, x, y, fgcolor, bgcolor, size)`: `text` of type `String`, `x` and `y` of type `Integer` or `Real`, `fgcolor`, `bgcolor`, and `size` of type `Integer`
- `Triangle(x0, y0, x1, y1, x2, y2, color)`: `x0`, `y0`, `x1`, `y1`, `x2`, and `y2` of type `Integer` or `Real`, and `color` of type `Integer`
- `Polygon(points, color)`: `points` is a list of containers of type `Point`, and `color` is an integer

- `Ellipse(x, y, a, b, color, filled)`: `x`, `y`, `a`, `b`, and `color` of type `Integer`, and `filled` of type `Bool`
- `FloodFill(x, y, color)`: `x` and `y` of type `Integer` or `Real`, and `color` of type `Integer`

This means that after a call to `initGraphics`, all containers whose names are listed above must have the parameters listed above. The attribute types are not enforced for the container definition to be correct, but the types given here are the types expected in the fields of containers passed as arguments to the graphics functions.

The `initGraphics` function may be called when some of the types described above have already been defined, but if the already-defined types have a different definition from what is expected here, an error will be raised.

Here is a more explicit description of the purpose of the various fields of these objects.

- The fields `x`, `y`, `x0`, `y0`, ... represent pixel coordinates
- The fields `a` and `b` of ellipses correspond respectively to the horizontal radius and the vertical radius of the ellipse
- The fields `color`, `fgcolor`, and `bgcolor` are colors. `fgcolor` is the color of the letters when drawing text, and `bgcolor` is the background fill color around each letter.
- The `filled` field indicates whether the shape should be drawn by filling its outline. Note that triangles are automatically filled, and polygons are not.
- The `width` and `height` fields represent respectively the width and height of objects. For example, to draw a rectangle, `width` and `height` correspond to its width and height. The `x` and `y` fields of a rectangle are the coordinates of the upper-right corner of the rectangle to be drawn.
- `radius` is the radius of the circle
- `size` is the size of drawn characters. A `size` parameter of 1 corresponds to basic-size text. For `size` = 2, the height of letters is doubled. For `size` = 3, both the height and width of letters are doubled. For `size` = 4, the height is tripled and the width is doubled. The logic is similar for subsequent values. Odd `size` values correspond to letters uniformly scaled by a ratio, and even values correspond to an intermediate size where only the height of letters has been stretched. You need not understand all of this; just understand that the larger `size` is, the larger the text.

Now let us look at the purpose of all these objects.

7.1.1 - The Screen

The screen of the TI-83 Premium CE / Edition Python (or TI-84 Plus CE) is a rectangle 320 pixels wide and 240 pixels tall.

The coordinate system places the origin (x=0, y=0) at the top-left. Thus, the pixel at the bottom-left corner has coordinates (x=0, y=239), the pixel at the top-right corner has coordinates (x=319, y=0), and the pixel at the bottom-right corner has coordinates (x=319, y=239).

As you may have noticed in the object definitions in the previous section, colors are integers between 0 and 255.

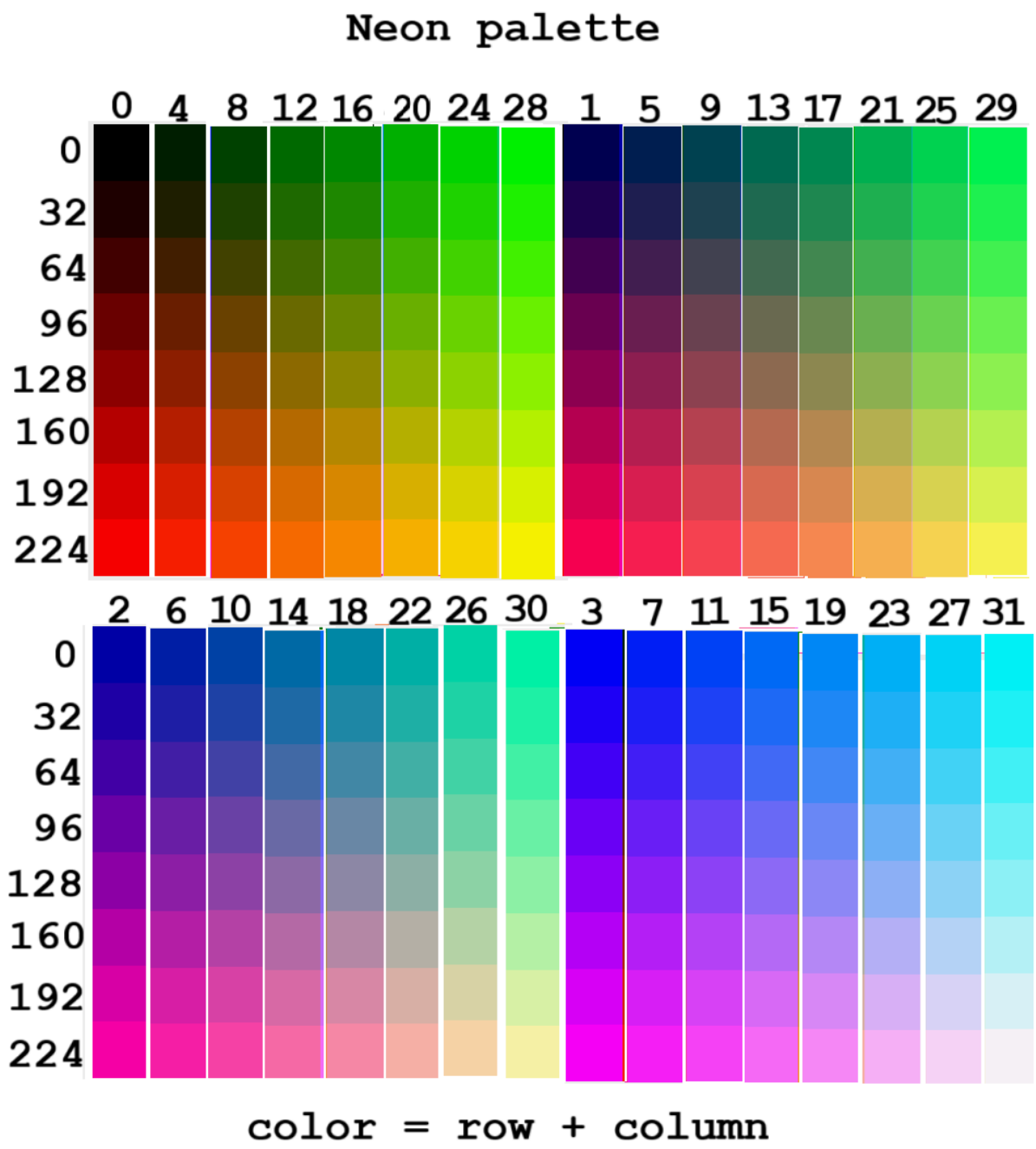
Each color is encoded in RGB format in 1 byte (8 bits) as follows:
- 3 bits for red
- 3 bits for green
- 2 bits for blue

Thus, there are 8 shades of red and green, and 4 shades of blue, and all combinations give exactly 256 different colors.

Because it is not practical to calculate `r * 32 + g * 4 + b` every time a color is needed, there is an `rgb` function to convert colors from RGB format to Neon's single-byte format.

For drawing text (the `Text` object), color 255 represents the transparent color.

Here is a representation of all colors accessible in Neon:



The `rgb` function

The colors available for graphics are integers between 0 and 255. This representation is not very convenient, so the `rgb` function maps an RGB color to its Neon equivalent. This function necessarily makes approximations, since it is impossible to store as many colors as 256×256×256 in values from 0 to 255.

The `rgb` function takes three integers between 0 and 255 as arguments, corresponding respectively to the red, green, and blue levels.

The `draw` function

The `draw` function is the main function for drawing things on screen. This function displays any number of objects at once.

The number and types of arguments expected by this function are completely free; anything can be passed to it. If an object passed to `draw` is directly drawable (a container of type Rect, Line, etc.), it will be displayed on screen. If an object passed to `draw` is a list or a non-drawable container, all drawable objects it contains (at any depth within the object) will be displayed. Objects of type `Integer`, `Bool`, etc. are ignored. If an object passed to `draw` contains a function (built-in or user-defined), the function will be executed.

In general, the only thing to remember about using this function is: `draw` will always try to draw whatever is drawable, no matter how deep it must search within the given arguments.

When drawing multiple objects on screen, draw order is often important. Some objects must appear above others, and others at the very back. The `draw` function always draws objects from left to right.

Example:
Suppose we want to code a boat racing game. We would use a `boats` list in which all the boats are stored.

Let us use for example this container type to represent boats: `Boat(speed, name, shape)`. The `speed` field is a number indicating the boat's speed and `name` contains the player's name.

The `shape` field contains a list of drawable objects for drawing the boat (for example: a filled rectangle for the outline, a triangle for the bow, and a few lines to show the wake behind the boat).

In a game using such objects, drawing all boats is simply a matter of calling `draw(boats)`.

The behavior of most objects when drawn is clear enough: `Circle` draws a circle, `Rect` draws a rectangle, `Line` draws a line, etc.

The `FloodFill` object performs a fill of a certain area of the screen at the specified coordinates.

The `setPixel` function

The `setPixel` function allows directly changing the color of a pixel.

`setPixel(x, y, color)` sets the color of the pixel at abscissa `x` and ordinate `y` to `color`.

- `x` must be of type `Integer` or `Real`
- `y` must be of type `Integer` or `Real`
- `color` must be of type `Integer`

The `getPixel` function

The `getPixel` function allows retrieving the color of a pixel. `getPixel(x, y)` returns the color of the pixel at abscissa `x` and ordinate `y`. `x` and `y` must be of type `Integer` or `Real`.

The `setTextTransparentColor` function

When drawing a `Text` object, you must specify `fgcolor` and `bgcolor`, the foreground and background colors of the text.

The `setTextTransparentColor` function allows specifying a color that will be used *as transparent*. For example, if you call `setTextTransparentColor` with a certain color, and then use that color as the `bgcolor` parameter, the text will not be highlighted.

By default, the color used for transparency is color `255`.

The `getTextWidth` function

This function computes the width in pixels of a given `Text` object, taking the `size` parameter into account.

The `menu` function

This function displays an interactive menu for choosing between different options.

It takes three parameters:
- The menu title (a string)
- The list of items to display (a list of strings)
- A string to display in case the item list is empty

The `menu` function returns the chosen item (a string), or `None` if the user did not choose anything.

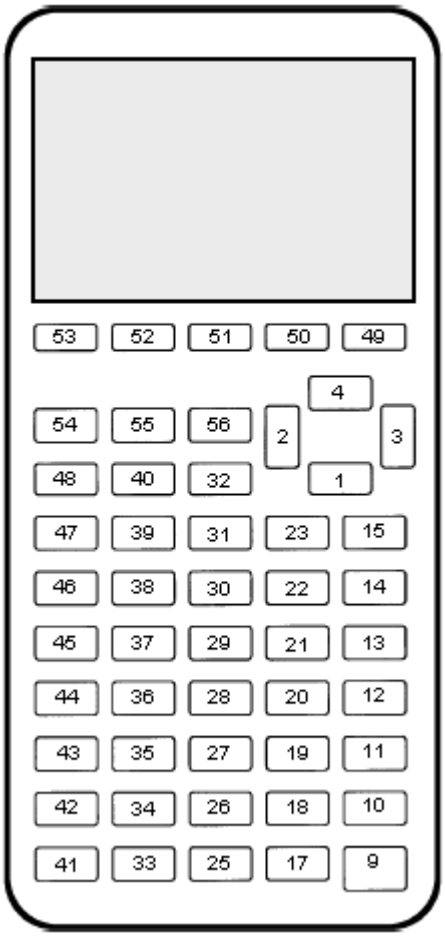
7.1.1 - The Keyboard

Keyboard access is managed by a single function: the `getKey` function, which takes no arguments.

This function returns an integer corresponding to the last key pressed, or zero if no key has been pressed since the last call to `getKey`.

Each key on the calculator has a unique code to identify it. The code associated with each key is shown on the diagram below.

The ON key (41) cannot be used with `getKey`. Pressing this key raises the **`KeyboardInterrupt`** error.



Conclusion

This documentation aims to be an exhaustive description of the features of the Neon programming language. If you think information is missing, information is incorrect, for any comment/question, or to report a bug, feel free to join the Neon Discord server: `https://discord.gg/wkBdK35w2a`.

You can also contact me by email at `contact@langage-neon.raphaael.fr`.