The Slow Programming Language

E. Arner

Table of Contents:

Preface

Last Updated: October 16th, 2021
Edition: 1.1 Beta – *Not For Production Usage*

Author:
- Email: arner@usa.com

The Slow Programming Language is an interpreted programming language implemented within the Python programming language—hence its name, due to the speed deficits that it experiences as a result—that is high-level and can compare to other popular programming languages. The Slow Programming Language, albeit it being capable, is not meant to be taken seriously, due to its speed deficits and its inability to be properly maintained by one person; the Slow Programming Language only exists to showcase one of the many ways an interpreted programming language can be made, in addition for it being a project.

This document will showcase the features of the Slow Programming Language, the keywords and the arguments which they require, operators, syntax, and the standard library of the Slow Programming Language. If you wish to use the Slow Programming Language in any meaningful manner, then the standard library description written in this document will be of uttermost importance to you.

The Slow Programming Language is an object-oriented programming language which employs the usage of classes to provide the framework necessary to achieve things easily and with an understandable level of abstraction. The Slow Programming language wishes to provide an easy to read syntax but a syntax which is not too verbose, where code can be reused wherever possible. Therefore, the language will automatically cast types when necessary.

The Slow language is evidently based—one way or another—on the following other, more widely used programming languages:

- Java – *new* keyword, camel case, capitalized type names, emphasis on object-oriented nature, emphasis on cross-platform ability.
- Rust – Type notation (e.g., *x: T*, or *x() → T*)
- C++ – *public* and *private* fields, also operator overloading for most—but not all—operators.
- Python – *pass* keyword, as well as array and dictionary notation (e.g., *[…]* and *{ … }*), pass by assignment.

Why is this document not written in Google Docs?

- Due to wanting to do offline work, this document has been made as a document on LibreOffice, which is an abundant office suite found on many systems—including Linux.

Variables

Variables may be defined using the *let* keyword, where it is required that they be defined and initialized before their reference, unlike Python. Characteristics of these variables may be set by placing the words *const* or *castable* (mutually exclusive!) before the name of the variable being defined within the *let* instruction, separated by a space.

An explicit type, while not mandatory, may be given to a variable using Rust notation, where a colon will separate the type name from the variable name—with or without a space—like so: → *name: type*. Multiple acceptable types may be given by enclosing the many types in a *List* literal fashion; castable types may be given multiple options, where they are in order of priority.

For example,

let const x: String = "Hello, world!"

- This will make a constant—meaning that it cannot be changed by threat of error—variable with the strict type of *String* containing the contents *"Hello, World!"* which will remain forever constant due to the constraints.

let castable x: String = 10

- This will create a variable which has permission to cast to whatever types support the variable value being casted. In this case, x wishes to have a type of *String* but is being set to a value of *10* which is of type *int*; therefore, the *String* class will cast the integer it receives into a *String* and return the new instance.
- If a variable is fed a type that it does not allow—and there is no *castable* modifier—then the Slow Programming Language will error.
- *castable* requires a variable with a type. The Slow Programming Language will yield an error, *TypeException*, if that quality is not met.

Variables exist within a scope that they are in. Global variables take precedence, where the order of precedence follows the stack of blocks that have been appended throughout the program's life. A block is defined as any section of code which has its own scope and is terminated by a corresponding *end* block, **UNLESS OTHERWISE SPECIFIED!**

Variables are also passed by assignment, meaning that their values are not copied on assignment, but rather they merely become references to that value in memory. Slow is like this due to the fact that it is based on Python, meaning that this is quite trivial to implement—it's also nice, but frankly, it would be nice if it were explicit, like C.

Variables have a maximum length of 128 characters and must obey the REGEX rule of **[a-zA-Z0-9-_]**. A variable may not also start with a number, as those are served for a numerical literal. Variables prefixed with tilde (~) are reserved for special uses within the Slow Programming Language—you may not make variables with that character, as implied above through the REGEX rule.

There are special variables throughout the runtime of the Slow Programming Language. A table of these below should explain them:

~line	Returns the *Int* line number currently being executed.
~file	Returns the *String* filename of the script currently being executed.
~dir	Returns the *String* directory name of the script currently being executed.
~args	A *List* of the command-line arguments passed to the interpreter.
~env	A *List* of all environment variables passed to the program.

In general, variable names should be in snake case—meaning lowercase with underscores separating the words—and they should be concise and/or condensed. For example:

→ *apple_tree*

Variable names cannot be redefined within scope, which is quite strange considering the fact that the Slow Programming Language is an interpreted language—which we try to obscure, to a certain extent, which will be evident throughout the course of this relatively long document. If a variable already exists while a new one of that name is trying to be made, a *RedefinitionException* will be thrown and the interpreter will complain.

Operators

Due to the fact that the Slow Programming Language is an object-oriented programming language, operators typically correspond to special methods within the data types that they are operating on. For instance, when I am comparing for two strings to be equal via the == operator, internally the ~equals() method is being called in order to compare them. Below will be a chart of all operators that exist within the Slow Programming Language. Some operators are not elligible for operator overloading.

+	Add
-	Subtract
/	Divide
*	Multiply
^	Exponent / Radical
=+	Compound addition (equivalent—but also more efficient—to $x = x + n$)
-=	Compound subtraction (equivalent to $x = x - n$)
/=	Compound division (equivalent to $x = x / n$)
*=	Compound multiplication (equivalent to $x = x * n$)
*^	Compound exponentiation (equivalent to $x = x \wedge n$)
%	Modulus—find the remainder of
&&	Logical and
||	Logical or
B&	Bitwise AND
B|	Bitwise OR
B^	Bitwise XOR
B~	Bitwise Compliment Operator
B>>	Bitwise left shift
B<<	Bitwise right shift
!	Logical not (invert boolean)
?	Ternary. *condition ? true_result : false_result*
(…)	Precedence operator
(T)	Casting operator
…	Varadiac arguments
,	Execute multiple instructions on same line, comma-delimited.

[…]: T	Create a list. If one wishes, they may add a colon and provide the type *T* of the list. This is useful for enumerations (see *List* for more information).
{ … }	Create a table
.	Method or namespace access operator
sizeof	Return length of object
instanceof	Is object a of type b? Returns *Bool*.
indexof	Return the index of the iterator
new	Initialize and create a new instance of an object.
copy	Instead of passing by reference, make a copy!
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to
===	Instances of the object are completely the same.
!==	Instances of the object are absolutely not the same.
~	Returns whether something is in an object.
!~	Returns whether something is **not** in an object.
=>	Lambda expression. See more in *Functions/Lambdas*.

All operators pertaining to arithmetic and mathematical operations have their order of precedence following that of PEMDAS—if you do not remember:

1. Parentheses (which are absolutely highest on precedence even within the context of other non-mathematical operators as well)
2. Exponents (and radicals)
3. Multiplication; division and modulus
4. Addition and subtraction

Null, Void, and Noent

The concept of *null* and *void* exists in many other programming languages due to the fact that they are useful for describing the absence of something; however, a feature that is not ubiquitious is the concept of the *noent* type, which seeks to describe the absence not of a value, but of an item. To characterize *null*, *void*, and *noent*, a nice chart below may help to explain it:

Type:	Description:
null	Describes the absence of value. If something returns *null*, it holds nothing and is nothing. You cannot reference members of something that holds the value of *null*, lest *NullPointerException* will be thrown.
void	Describes the inability for a function, operator, or expression to return a value. It does not yield a value. You cannot use the result of *void* for anything, lest a *SyntaxException* will be thrown.
noent	Describes the lack of an entry for that thing. If we wish to apply a function to a dataset by mapping, returning *noent* would cause that value to not exist anymore within the returned dataset.

Literals:

- The literal *null* will return a value containing *null*, which may be applied to any variable with any type, without changing that type.
- The literal *void* will signify to the *return* keyword that the function does not return any value. If an operator returns *void*, that means that its return value may not be used.
- The literal *noent* will signify to *List* & *Table* manipulators that the value should have no entry if a specific condition is met.

Functions

Functions within the Slow Programming Language are defined using the function keyword, where the name of the function is next, separated by a space; onwards, the arguments of the function may be defined by a comma-delimited list inside parentheses. All functions must end with parentheses, as it keeps the syntactical structure consistent within the language.

The format for arguments is the same as they are for variables and/or the let keyword. Function arguments may be given *const* or *castable* qualifiers, and function arguments may bear strict types which may or may not be automatically casted given the aforementioned qualifiers passed to it.

Functions also employ the useful concept known as keyword arguments, where arguments may be set to a default value within a function by making them equal to a default value, but also allowing for them to be changed using a similar mechanism while calling the function.

If a function wishes to differ how it behaves on the basis of the types and numbers of its arguments, the function may overload—to allow different code based on the aforementioned condition—itself. The function merely must have the same name, but with different typed and numbered arguments. The standard library prefers overloading to keyword arguments when expanding functionality to functions, but this specification hereby declares no best solution—it is situational.

Finally, there are arguments which can be called *varadiac arguments*, being variable in number. Varadiac arguments in the Slow Programming Language act similarly to the ones found in C, being defined by '…' within the function arguments. Varadiac arguments, then, may be accessed using the *~vargs* variable which will be inside a function with varadiac arguments which will be a general *List* object.

A return type, while not required, is a possibility to enforce within the Slow Programming Language. If there is no return type specified, then the function may return anything it wishes to—which may be dangerous! If the return keyword is not used to return anything within a function, the function will return null. If you wish to specify that the function returns nothing and that its return type cannot be used, have it return void. To enforce a return type, use an arrow -> after the parentheses and then provide the return type.

A function's name is limited to a maximum of 64 characters and must obey the strict REGEX rule of **[a-zA-Z0-9_-~]**. A function's name may not begin with a numerical value. A function may not redefine another function that is within the same scope.

→

An example function, in the Slow Programming Language:

```
function doSomething(const x, castable y: String, z = “Hello”, … ) -> String
        print(x)
        print(y)
        print(z)
        print(~vargs[0])

        return x + y + z + ~vargs[0]
end
```

In order to call the aforementioned function,

doSomething(“Constant string”, <Castable bytes>, z = “Changed default keyword”, “First”, “Ignored”)

- *y* is casted from *Bytes* to *String*, automatically. If *y* did not bear the *castable* modifier, then the interpreter would yield an error.
- *y* also showcases a *Bytes* literal, where text may be represented as *Bytes* if surrounded with angle brackets.
- *z* is a keyword argument which has had its value changed.
- “First” is passed as a varadiac argument. “Ignored” is not used in the function, because the function only uses index *0* from the varadiac arguments.

When functions are non-static methods of a class, the class instance will always be passed as the variable this, albeit the scope of the class is generally accessible without this. this is still useful, however, for passing the instance of the class to other parts of the program and for removing ambiguity when referencing data.

Functions should be named in accordance with the camel case, where the initial word is not capitalized, though every subsequent word has its first letter capitalized, like so: thisIsAFunction().

Special methods of classes will be described in the *Objects/Classes* section.

You may store a function as a variable and then call it later, thereby implementing the concept known as ‘callback functions.’ A function when evaluated will have the type of *Function*, which is an object as is everything.

Functions should be in the style of camel case, where the first word is lowercase but all subsequent words are capitalized with no spaces spearating them, like so:

→ *doStuff()*

Anonymous Functions

An anonymous function can be described as a function literal, where it returns an expression containing that of a callable function, but without implicitly storing it as a variable somewhere. An anonymous function is quite identical to a normal function, with the exception that it bears no name. An anonymous function may be created using the *anonymous* keyword, then with the arguments of the anonymous function enclosed in parentheses, per standard. For example:

```
anonymous(x)
        print(x)
end
```

Lambdas

A lambda is a type of anonymous function, one which implicitly returns whatever operation is contained inside of it—for this reason, a lambda cannot contain multiple statements or lines of code, for there is no way to determine the expression to return—and is more like an algebraic function in that regard. A lambda may be created, in accordance to many other programming languages, with the => operator, where the arguments are on the left hand side and the return expression is on the right hand side; a lambda bearing no left side or right side is invalid and will error. For example:

```
→ x => x^2
```

- This function is quadratic, squaring whatever is put into it (if it is an integer or overloads the power operator); it will automatically return the result of the operation done inside of it.

Lambdas are quite useful in functional programming processes, where a defined function is not often needed to enact a certain function rule across elements of a *List* or *Table.* For this reason, they are very mathematical and algebraic in their abstract nature. Lambdas will be explained in practice with *List*s in the respective section on the *List* object in Slow's standard library.

Objects

In recognition of the fact that the Slow Programming Language is an object-oriented programming language, objects are the building block of the language, for most things attempt to be that of an object in this language. There are two distinct, albeit oxymoronically related, types of objects within the language: classes and enumerations.

Objects are limited to a REGEX rule of **[a-zA-Z0-9-_]** and a character limit of 64. Object names should be in pascal case, where each word is not separated by a space and has their first letters capitalized, like so: *ThisIsAnObject*.

Classes

Classes contain related data together along with functions to manipulate/utilize that data in order to achieve a result, in an abstracted manner. Classes in the Slow Programming Language attempt to model those found in C++, Java, and Python.

In order to define a class,

```
class MyObject
	let x: String = null

	function ~init(x)
		this.x = x
	end
end
```

A class has its constructor to be the name of *~init* where it will do all that is required for setting up the class when a new instance of it is made. It is recommended to define all of the variables that will be used throughout the class at the top of the program, where the constructor will set them to the appropriate values.

Unlike Python, the reference to the current class instance does not have to be passed as an argument to every non-static function—that is quite annoying and is a feature that, quite frankly, wastes precious time in such a concise language like Python.

Class inheritance, where an inferior class may inherit methods and variables from a superior class, is implemented using the *inherits* keyword, where the classes of inheritance are given by a comma-delimited (no comma needed for a single class inheritance) list, inherited by left-to-right priority.

→

class Child inherits Grandparent, Parent
    …
end

A situation will emerge where you may wish to call a function from one of the parent classes. If that situation arises, special variables prefixed with @ will be available which will allow you to access the methods of the parent classes, where the name will be that of the prefix and of the parent class you wish to access. To call the parent's constructor, which will operate within the scope of the child class:

→ @parentClass.~init()

Class visibility may be set downwards—until the issuance of another keyword—by one of the following keywords: *public*, *private*, and *static*. Class visibility is automatically set to *public* until further issuance of the aforementioned keywords. The *public* section of a class has all members accessible to everyone; the *private* section of a class has their members only accessible to the class and its children (if applicable); finally, the *static* section of a class has their members not dependent or unique to any specific instance, but across all instances and references to the class—a change to a static variable will affect all subsequent instances and its behavior.

Static methods and members may, as aforementioned, be made using the *static* keyword to set everything downward to a static member. Slow does not enjoy the verbosity given by languages such as Java, and the Slow Programming Language does not like the confusing and troublesome nature of Python with regards to class visibility.

Unlike Java and JavaScript, Slow employs the *extremely* useful nature of operator overloading; it additionally makes them understandable, unlike C++. However, the extent to which you can overload is limited, as to not be like Python in that regard. Below is a table of all of the special methods—those prefixed with ~–which may correspond to the various operators in the Slow Programming Language.

~add(x)	+	Add another instance of the class and return the result.
~sub(x)	-	Subtract another instance of the class and return the result.
~mul(x)	*	Multiply another instance of the class and return the result.
~div(x)	/	Divide another instance of the class and result the result
~mod(x)	%	Use modulus against another instance of the class and return the result.
~pow(x)	^	Raise this instance to the power of another instance.
~cast(x): T	(x)	Convert *x* of type *T* to an instance of this class and return it.
~index(x)	[x]	Get the index *x*. Useful for tables and arrays.

~setindex(x, y)	[x] = y	Set the index *x* to *y*
~sizeof	sizeof	Get the count of the instance. Useful for tables and arrays.
~del(x)	del x	Delete an index of an array or table.
~bool()	N/A	When evaluated under an *if* or *else if*, return a bool indicating the state of the instance—*is it ready for reading* or *has it hit eof*, for instance.
~iter()	N/A	Begins an iteration; the existence of this method will also signify that this object is iterable.
~next()	N/A	Obtains the next item in the object is that is being iterated over. Yields *EndOfIterationException* to signify that the end of the iteration has hit. Another call to *~iter* will reset the internal counter.
~lesser(x)	<	Returns whether instance is less than *x*
~greater(x)	>	Returns whether instance is greater than *x*
~lesserequal(x)	<=	Returns whether instance is less than or equal to *x*
~greaterequal(x)	>=	Returns whether instance is greater than or equal to *x*
~equal(x)	==	Returns whether instance is equal to *x*.
~in(x)	~	Is *x* in the instance? This is useful for list or table-like objects.
~copy()	copy	Upon issuance of the *copy* operator, return a copy. *This is for Python implementation purposes only.*

Enums

Enumerations are essentially options which can be given to functions or classes in order to operate differently. An argument or variable with the type of an enumeration has their scope and availability of values constrained, quite forcibly, to only the options that the enumeration has defined.

In order to define an enumeration:

enum Enumeration

    Option1

    Option2

    Option3

    …

end

- Enumeration options should be in pascal case, like so: *AnOption*.

Enumeration *Enumeration* will have three options to select from: *Option1*, *Option2*, and *Option3*. As the name implies, each option will count up one, starting from zero, yielding the values: 0, 1, and 2. Unlike many other programming languages, each option is separated by a newline instead of delimiting by comma—where it is not valid to do so in the Slow Programming Language.

An enumeration's options is restricted to **[a-zA-Z0-9]**, with a maximum of 32 characters for their option names.

Keyword commands

Keywords are a list of primitive instructions that the Slow Programming Language is built upon. Another name for keywords may be the word ‘built-ins’ as programming languages such as Python would call them. Consider this to be the miscellaneous section of this specification.

_out *text* – (Debugging purposes only) Prints out *text* which is raw and has no type.

- This should not be used in lieu of *print*! This is only for debugging purposes.

exit *(optional) code* – Exit; additionally, you may provide an exit code.

- The exit code must be from 0 – 255.

Parallelism and Multithreading:

parallel *statement* – Execute *statement* parallely, without blocking other sections of the program which is useful for IO and networking—if Slow will ever do that.

end – End the current block of code. This is analoguous to a *}* in C-like programming languages.

Bool

A boolean is a data type that represents a true or false statement. Booleans are useful for taking a particular action given a specific condition. A boolean, in the Slow Programming Language, is referred to as *Bool*.

true – Returns bool of value *true*
false – Returns bool of value *false*

- Note: fixed Python's improper capitalization of boolean literals.

Certain operators will return a boolean, namely those which are trying to verify if a certain quality or condition is met.

0, 0.0, and "0" may be casted to a bool, with the value *false*.
1, 1.0, and "1" may be casted to a bool, with the value *true*.

- Note: other types should implement vice versa themselves.
  - *Bool* should be able to be casted to its *String* representation.

A *Bool* may not be checked with equality operators, for that is bad practice. It will not implement any of those operators. Code that may look like this, *x == True*, will not exist within the Slow Programming Language for fairly evident reasons.

String

The *String* datatype is a fully object-oriented representation of text within the Slow Programming Language. The *String* object allows you to preform easy manipulations of text in an abstracted manner. Most things should have the ability to be casted to a *String*.

A *String* may be created using double or single quotes, like so: “Hello world” or ‘Hello world.’

A *String* may be formatted by placing a dollar sign and curly braces inside of it and returning an expression that would return a *String* or something that may be casted to a String. For example: “Hello, ${user}” → “Hello, admin”

Methods and members:

Members:	Description:
length: Int	Returns how long the string is. *sizeof* will also return this.
offset: Int	If this *String* was returned as a subset of another *String*, what offset is it from the original *String*?
end: Int	If this *String* was returned as a subset of another *String*, where does it end relative to the original *String*?
eof: Bool	If this *String* was returned as a result of a file reading operation, is this the last *String* you will read—*has end-of-file occurred*?
split(delimiter: String, escape: String = null) -> List	Split a string into tokens which is a *List*. You may also provide an escape string—such as quotations—so that anything in them are ignored by the delimiter.
reverse() -> String	Reverses the string and returns the reversed string.
encode() -> Bytes	Encodes the *String* into *Bytes.* Casting the *String* to *Bytes* will also accomplish this.
strip(what: *String*) -> *String*	Strip certain characters from both ends of the *String*.
strip() -> *String*	Strip whitespace characters (\r\n, \n, space, tab, etc) from both ends of the *String*.
lstrip(what: *String*) -> *String*	Strip certain characters from the left side of the *String* only.
lstrip() -> *String*	Strip whitespace characters from the left side of

	the *String* only.
rstrip(what: *String*) -> *String*	Strip certain characters from the right side of the *String* only.
rstrip() -> *String*	Strip whitespace characters from the right side of the *String* only.
capitalize() -> String	Return a capitalized *String*
lowercase() -> String	Return a lowercased *String*
join(list: List) -> String	Using a *String* as the joining string for each element of the array, join elements of an array (which can be casted to a *String*), and return a *String* containing the result.
find(substr: String) → String	Find a substring within a parent *String*. Return a new *String* containing the original String but offset as to start at that substring (similar to *strstr* in C). If you want the offset, access the *offset* member of the returned *String*.
format(…) -> String	Format a *String* using C# String formatting notation. • Example: *"{0}".format(thisText, …)*
format(values: *Table*) -> String	Format a *String* using Pythonic String formatting, but with a table: • Example: "{name}: {age}".format({"name": "John", "age": 26}) • Useful for formatting files or large blocks of text. For both types of *format*, the dollar-sign prefix ($) can be escaped with \$.

The following operators and special methods may be used on a *String*:

Method:	Operator (if applicable):	Description:
~add(x) -> *String*	+	Concatenates two *String*s.
~in(x) -> *Bool*	~	Finds if *x* is a substring of the parent *String*.
~sizeof() -> *Int*	*sizeof*	Returns the length of the *String*.
~equals(x) -> *Bool*	==	Returns whether two *String*s are equal

Like many other programming languages, Slow's *String* class includes the ability to reference special characters, such as newlines or carriage returns, with escape characters. Slow will largely reference the style used in C (e.g., "\n"). Here is a list of all characters and what they do:

Character:	Description:
\n	Newline character. May or may not be sufficient—Windows requires \r\n.
\r	Carriage return. May or may not be sufficient—Linux doesn't require it.
\l	Universal newline character. This shall be preferred when it is not necessary to explicitly have the two previous characters.
\$	Format escape.
\a	Alert. If in a compatible terminal, it should ring a bell or alert the user somehow.
\b	Backspace character.
\e	Terminal escape code.
\f	Form feed page break.
\t	Horizontal tab
\v	Vertical tab
\\	Backslash
\'	Apostrophe
\"	Quotation marks
\xHH	Hexadecimal byte.

Int, Decimal, and Complex

Int

An *Int* is a datatype which holds any whole number ranging from negative to positive. An *Int* has no bearing on length and will accommodate the maximum length of word on whatever system the Slow interpreter is running on. On x86_64, *Int* will have a maximum length of 2^{64}; conversely, on i386, *Int* will have a maximum length of 2^{32}.

An *Int* may be created when writing a non-decimal number with or without a negative sign. An *Int* may additionally be created in a different base than base 10. The prefixes for the following bases are as follows:

- 0x: for hexadecimal numbers (e.g., 0xFFFF)
- 0: for octal numbers (e.g., 0755)
- 0b: for binary numbers (e.g., 0b11110000)

To help with reading numbers in source code, an underscore may be added in between the digits of the *Int*. For instance: 350_000_000 is identical to 350000000, albeit much easier to read.

Int also implements mathematical/programmatic concepts such as infinity and NaN. The concept of infinity shall be available with the literal Infinity, where a minus sign may be added to negate it to negative infinity. The concept of NaN (not a number) shall be available with the literal NaN. Consequently, only the *Int* object will bear the following members in regards to the aforementioned items:

Members:	Description:
infinity: Bool	Is this number infinity? Default: *false*
ninfinity: Bool	Is this number negative infinity? Default: *false* • Note: *infinity* and *ninfinity* are mutually exclusive.
nan: Bool	Is this number *NaN*? Default: *false*. • Note: *NaN* is mutually exclusive to any of the infinities.

Decimal

In the case that the number in question contains fractional parts, then *Decimal* will be the appropriate datatype to use. The length, as with *Int*, will be determined based off of the system that the Slow interpreter is running from.

Decimal literals are slightly different than that of Int: in order to create a Decimal, one must add a decimal point to the number or add ‘D’ as a suffix; for example:

- 5.0
- 2.3
- 10D

A mathematical equation will return a Decimal if:

- One or more of the numbers involved is a Decimal
- If division is done, to be safe.

If a Decimal has no fractional part, it may be casted to an Int.

A decimal will have its own unique methods, mainly operating on the decimal portion of its result. Here they are:

Method:	Description:
ceil() -> *Int*	Round the decimal up to the largest integer: • 1.9 will yield 2 • 1.2 will yield 2
floor() -> *Int*	Round the decimal down to the largest integer: • 1.999 will yield 1 • 1.0001 will yield 1 This function essentially is the opposite to modulus, returning the number without a fraction, while modulus returns the amount left over.

Complex

A *Complex* number is a number containing a real and imaginary part to it. Generally speaking, this is most likely not going to be used often, considering that Slow—being slow—will not be used for mathematical applications that would require such a datatype; therefore, the standard library shall not include functions which deal with complex numbers, opting to store them in a separate package named *Cmath*.

In order to create a complex number, write the real number +/- imaginary number, suffixed with *i*. For example:

- 10 + 2i
- 0 - i

- 90 + 0i
  - Even though this is a real integer, it is still under the *Complex* datatype.

Methods

All of the types listed above will bear the following operators and methods to implement them, where *T* represents either *Int*, *Decimal*, or *Complesx*:

Methods:	Operators:	Description:
~add(x: *T*) -> *T*	+	Adds
~sub(x: *T*) -> *T*	-	Subtracts
~div(x: *T*) -> *T*	/	Divides
~mod(x: *T*) -> *Int*	%	Modulus—find the remainder.
~mul(x: *T*) -> *T*	*	Multiplies
~pow(x: *T*) -> *T*	^	Raises to the power of
~sizeof() -> *Int*	*sizeof*	Returns the size, in bytes, of the number in question.
~equals(x) -> *Bool*	==	Is *x* equal to this number?
~greater(x) -> *Bool*	>	Is *x* greater than this number?
~lesser(x) -> *Bool*	<	Is *x* less than this number?
~greaterequal(x) -> *Bool*	>=	Is *x* greater than or equal to this number?
~lesserequal(x) -> Bool	<=	Is *x* less than or equal to this number?

Bytes

Raw data in the Slow Programming Language is represented with the *Bytes* datatype/object. In a similar fashion to a *String*, *Bytes* merely contains an array of bytes, though is differentiated so that each one may exercise their specialities in their respective purview.

In order to create a *Bytes* literal using characters, enclose the text inside angle brackets, like so:

- <Hello>
- <This is a message.>

The more efficient approach would be to create a *String*, only to then cast it to a *Bytes*, given that it can be encoded properly. It would be more convenient, due to the fact that you can format a *String* with such elegance.

Reading binary data from files will return a *Bytes* object.

A *Bytes* object will have the following members and methods:

Members:	Description:
length: Int	The size of the *Bytes* object (how many bytes are contained within it). *sizeof* will return this.
offset: Int	If this *Bytes* object was created as a subset of a larger *Bytes* object, what is the offset from the larger *Bytes* object, if applicable?
end: Int	Given the same situation as above, when it does it end, relative to the larger, parent *Bytes* object?
eof: Bool	Has *eof* occurred yet? By default: *false*.

Below are a list of special methods—and if applicable—with their corresponding operators.

Special method:	Operator (if applicable):	Description:
~bool() -> Bool	N/A	Returns whether the *Bytes* object has not reached *eof* yet as a *Bool*.
~sizeof() -> Int	*sizeof*	Returns the length of the *Bytes* object.
~equals(x: Bytes) -> Bool	==	Returns whether *x*—the other object being compared, whether automatically casted or not—is identical to the bytes of this object.
~in(x: Bytes) -> Bool	~	Returns whether *x* is a subset or is included in the parent *Bytes* object.

List

A *List* is used for the purposes of storing multiple variables and/or values under a single container, noting the length of it, and possessing the ability to dynamically increase the container size upon new items being added to it. A *List* in the Slow Programming Language is almost identical in concept and function to that of a list in Python—no strict bearing on type and *extremely* easy to use.

As explained above, a list may be initialized with hard braces, like so:

[item1, item2, item3, …]

Additionally, a list may be given a type suggestion for enumerations, where every value inside will be within the scope of the enumeration, lacking the need to type the enumeration name:

```
enum Options
        optionOne
        optionTwo
        optionThree
end

[optionOne, optionTwo, optionThree]: Options
```

- Only *List*s given enumerations as their index type as a literal will have this behavior. Other types shall not be permitted, yielding *TypeException* if provided.

A *List* is an iterable object, so it may be used within a *foreach* loop, like so:

```
foreach (item in listObject)
        doSomething(item)
end
```

You may obtain an item using the *[]* operator. Indices are numerical and cannot be anything else. A *List*'s index also begins at zero, as it does in many other programming languages. For example: *listObject[2]* will return the third element within the *List*.

The following methods and members will be available on a *List* object:

Member:	Description:
length: Int	The amount of elements in the *List*. *Analogous to ~sizeof(x).*
add(x) → void	Add *x* to the *List*.
insert(ind: Int, x) → void	Insert *x* at index *ind*, moving everything over. *This gives the list stack-like properties when ind = 0.*

push(x) -> void	Push an element to the front of the stack-like *List.*
remove(x) → void	Remove all instances of *x* in the *List*.
index(x) → Int	Return the index number of *x*, if it is in the *List*.
reverse() -> List	Reverse the current *List* and return the result.
sort() -> List	Sort—using the best algorithm—the *List* and return the result.
right(x: Int) → List	Offset *x* indices to the right and return the result. This is indentical to *x[y:]* in Python.

You may delete an element using the *del* operator:

→ *del myList[2]*

The following operators and special methods, while some of them are obvious, will be implemented in the standard library's *List* object:

Method:	Operator:	Description:
~in(x) -> Bool	~	Returns whether *x* is in the *List*.
~del(x) -> void	*del*	Deletes *x* from the *List*. If index not found, yield *IndexException*.
~index(x)	[]	Obtains an index *x* from the *List* and returns the result.
~setindex(x, y)	[x] = y	Sets index *x* of the *List* to *y*.

Function rules

A function rule may be applied to a list by using the bracket operators and separating the rule and the the source list which the rule shall be applied to, using ->. For example, considering that *aList* contains odd numbers starting from 2:

→ *let newList: Int[] = [x != 2 ? 2*x : noent -> aList]*

- *x != 2 ? 2*x : noent* will only execute *2x* if *x* is not 2, where if *x* is 2, the result will not be added to the new list, making it non-existent. This is an excellent display of the ternary operator, as well.

Table

A *Table* is an object within the Slow Programming Language which associates one object to another. In other terms, it is analoguous to Python's dictionary object or C++'s std::map object. The format in which one would use to initialize a *Table* with values is that of JSON:

→ *{key1: value1, key2: value2, key3: value3, … }*

In order to obtain a value from the *Table*, use the *[]* operator like so:

→ *myTable["myKey"]*

You may wish to set values, which can be shown in the following example, using the *[]* operator:

→ *myTable["myKey"] = "myValue"*

The following members will be present on the *Table* object:

Members:	Description:
length: Int	Returns the amount of keys within the *Table*. *sizeof* will, most definitely, return this as well.
clear() -> void	Clear the *Table* of keys and values.
keys() -> List	Get the keys as if they were a *List*.
values() -> List	Get the values as if they were a *List*.
update(src: Table) -> void	Update the current *Table* with another *Table*, adding or replacing values from *src*.

In order to delete an item from the *Table*:

→ *del myTable["myKey"]*

The following operators and special methods will be available under the *Table* object:

Method:	Operator:	Description:
~in(x) -> Bool	~	Is *x* in the *Table*?
~index(x)	[]	Return index *x* from the *Table*. • If element *x* does not exist within the *Table*, *IndexException* will be yielded.
~setindex(x, y)	[x] = y	Set index *x* of *Table* to *y*.

		• If *x* does not exist, it will be created within the table.
~del(x)	*del*	Deletes element *x* from the *Table*.

JSON

A *Table* natively supports JSON, allowing the casting of a *String* containing JSON data to an instance of a *Table*—as well as vice versa. For instance, to make a *Table* with a *String* JSON object:

→ let x = (Table) “{‘a’: ‘b’}”

or

→ let castable x: Table = “{‘z’: ‘y’}”

Time

The keeping of time is crucial for the high-level applications of a high-level programming language; it is, however, quite tedious and lacks a concise standard, due to the formatting of time being complicated and quite difficult to remember. The Slow Programming Language implements the *Time* object to represent a date and time in a meaningful manner. You may initialize *Time* without a constructor to operate on the current date and time, or you may initialize *Time* with an *Int* describing the UNIX timestamp of the date and time:

→ *let time = new Time()* or *let time = new Time(462382372)*

The *Time* object will also have a *TimeFormats* enum associated with it, to help with formatting the time and date when converted—not casted—to a *String*, with comments to showcase the general output of the enumeration:

```
enum TimeFormats
	Weekday // Monday
	ShortenedWeekday // Mon
	WeekdayNumber // 2
	ZeroedDayNumber // 09
	DayNumber // 9

	ShortenedMonth // Jan
	Month // January
	ZeroedMonthNumber // 01
	MonthNumber // 1

	ZeroedYearNumber // 09
	YearNumber // 9
	FullYear // 2009

	ZeroedHour24 // 03:00
	Hour24 // 3:00
	ZeroedHour // 03:00
	Hour // 3:00

	Meridian // PM

	ZeroedMinute // 09
	Minute // 9

	ZeroedSecond // 02
	Second // 2
end
```

The *Time* object will also have the *TimeAmounts* class associated with it, which will contain constants useful for time, as will be made obvious below:

```
class TimeAmounts
	static

	// Below are a list of time units, in seconds, which are useful for time.

	let const Second = 1
	let const Minute = 60
	let const Hour = 3600
	let const Day = 86400
	let const Week = 604800
	let const Month = 2419200 // Approximate, for it is calculated as 4 weeks.
	let const Year = 31557600 // Approximate, for it is calculated as 365¼ days.
end
```

The methods and members which the *Time* object must implement:

Members:	Description:
timestamp: Int	The UNIX timestamp of the date and time encompassed.
date: *Bool*	Is this *Time* instance not only representing time, but a date? By default, *true*.
format(options: *TimeFormats[]*) -> *String*	Given an array of formats—there cannot be multiple instances of the same item—format the time into a *String* which is user-friendly. → *new Time().format([Month, ZeroedDay, FullYear, ZeroedHour, Meridian, ZeroedMinute, ZeroedSecond]: TimeFormats)*

The following operators and their methods are supported by the *Time* object, where arithmetical operations are done on the time contained within the *Time* object.

Members:	Operator:	Description:
~add(x) -> *Time*	+	Adds two instances of the *Time* object together and returns the sum.
~sub(x) -> Time	-	Subtracts two instances of the

		Time object and returns the difference.
~mul(x) -> Time	*	Multiplies two instances of the *Time* object and returns the product.
~div(x) -> Time	/	Divides two instances of the *Time* object and returns the quotient.
~sizeof() -> *Int*	*sizeof*	Time in seconds (timestamp).
~equals(x) -> Bool	==	Is *x* the same time as this time?
~greater(x) -> Bool	>	Is *x* further into the future than this time?
~lesser(x) -> Bool	<	Is *x* further into the past than this time?

Path

Path will preform operations that are useful for managing paths: you may wish to know whether a path exists, if it is relative, or you may wish to change/view the mode of the path. A path may be to a file or a directory, but it serves a far more general purpose. It is such that, perhaps, *Path* should have been a parent class for which the following *Directory* and *File* classes inherit rightfully from, though *Path* is just being used as a sanity check for the path which *Directory* and *File* may want. *Path* must be initialized with a *String* containing the Windows or UNIX path you wish to operate on:

→ *let path = new Path(“/path/to/directory_or_file”)*

- Note: UNIX path abbreviations, such as ~, work with the *Path* object.

If the path is incorrect or cannot be converted between path formats, a *PathException* will be thrown, which is useful as a sanity check for any other classes or functions which may require a path, but may not wish to manually check it via a *String*. It is **important** to note that a *String* path will automatically be casted to a *Path* where specified, thereby bypassing the verbosity of languages such as Java.

Here are the members and methods available within the *Path* object:

Members:	Description:
exists() -> Bool	Returns whether the path exists.
mode() -> Int	Returns the UNIX octal file mode of the path, if applicable.
basename() -> Path	Returns the basename—the path omitted from the file—of the *Path*.
dirname() -> Path	Returns the directory name, without the filename at the end.
accessed() -> Time	Returns a *Time* object containing the last access time of that *Path*, if it exists.
modified() -> Time	Returns a *Time* object containing when the path was last modified.
created() -> Time	Returns a *Time* object containing when the path was created.
real() -> Path	Return the real path, bypassing symbolic links.

The *Path* object supports the following operators and special methods:

Method:	Operator (if applicable):	Description:
~add(x)	+	Add paths, automatically appending the appropiate separator if applicable.
~sub(x)	-	Subtracts paths, automatically including the separator within the subtracting of paths if applicable.
~bool()	N/A	Returns whether the path exists, being analoguous to *exists()* which can be found as a member of *Path*.

File

In the Slow Programming Language, file I/O is handled through the *File* object. The *File* object must be initialized with the name of the file that is supposed to be worked on, along with the mode it should use in order to read/write to the file. It is recommended that operations with the *File* object be used in conjunction with a *with* block, so that instances are scoped and streams are automatically closed.

The *File* object will automatically convert path types to the system's native one, within its constructor.

The following modes are available, under *FileModes*,

enum FileModes
    Read
    Write
    Append
    ReadBinary
    WriteBinary
    AppendBinary
end

Here is a list of methods and functions available under the *File* object:

Methods:	Description:
exists() -> Bool	Returns whether the file exists.
open() -> void	Opens the file for reading or writing—*with* will automatically do this. *The default file mode will be used*.
open(mode: Int) -> void	Open the file for writing, and *if creating a new file*, set the UNIX octal mode automatically.
close() -> void	Close the file when you are done—*with* will automatically do this.
write(data: Bytes) -> Int	Writes *Bytes* to a file and returns the amount wrote. • Returns *OSException* if the file cannot be wrote to.

write(text: String) -> Int	Writes a *String* to a file and returns the amount wrote. • Returns *OSException* if the file cannot be wrote to.
read() -> String, *if in text mode.*	Reads the entire contents of the file into a *String*. • Complains with *OSException* if the file cannot be read.
read() -> Bytes, *if in binary mode.*	Reads the entire contents of the file into a *Bytes* object. • Throws *OSException* if the file cannot be read.
read(length: Int) -> Bytes, *if in binary mode.*	Returns *length* bytes of the file as *Bytes*. If *EOF* was hit, the *Bytes* length will be less than what was put into *read*. • Returns *OSException* if the file cannot be read.
readline() -> String, *if in text mode*	Returns a *String* line of the file, seeking to the next line, for the next *readline()* if applicable. Returns, as always, *EOF* if there are no more lines to read. • Returns *OSException* if the file cannot be read.
readline() -> Bytes, *if in binary mode*	Returns a *Bytes* line of the file, seeking to the next line, for the next *readline()* if applicable. Returns, as always, *EOF* if there are no more lines to read. • Returns *OSException* if the file cannot be read.
go(where: Int) -> void	Go to position (character) *where* in the file. • Going past the file size will fill the space with zeroes. • Raises *OSException* if the file is not seekable.
start() -> void	Go to the start of the file. • Raises *OSException* if the file is not seekable.
end() -> void	Go to the end of the file. • Raises *OSException* if the file is not seekable.
position() -> Int	Return the position of the file currently.
seekable() -> Bool	Returns whether the file is seekable—a special file may not be!
writable() -> Bool	Returns whether the file is writable.

readable() -> Bool	Returns whether the file is readable.
fileNumber () -> Int	Returns the file descriptor, if applicable. • Yields *OSException* if the *File* has no file descriptor.
close() -> void	Close the *File*, safely exiting.
isSpecial() -> Bool	Returns whether the file is special.

To make an addition to the regular methods of the *File* object, here are the special methods and their—if applicable—corresponding operators, along with what they do and what they might complain with:

Method:	Operator (if applicable):	Description:
~sizeof(x) -> Int	*sizeof*	Returns the length of the file encompassed by the *File* object. • If the file is not found, a *FileNotFoundException* will occur, stopping the program in its tracks.

stdout & stderr

As with most other languages, the terminal console is represented by files. The output of the terminal is represented by a *File* instance—which is global—called *stdout*. If one wished, they may write to it directly, albeit that is confusing and goes against what this specification desires.

Instead, one may opt to use the global *print* function, which has the express purpose of writing data to the *stdout* file—or your terminal. If you are printing an error message, it is best to use *stderr* to represent it, per standard; additionally, you may use the *eprint* function to accomplish that.

print and eprint will automatically append newlines to the end of their messages. If you do not wish for this functionality, please use printn or eprintn—print without newlines.

stdin

We must also read from the console at times, which is what *stdin* represents: a file open to—depending on your system—to file descriptor 0 (or /dev/stdin) with *Read* set. Again, it is confusing to merely read this as a normal file, so we have dedicated global functions for that.

You may use the *read()* function to read input from the terminal. If you wish to have a prompt along with the reading, you may overload a *String* into the *read*() function.

Directory

Directories are a type of file, but they must be dealt with in a different fashion. It is useful, for instance, to know of all of the files in a directory or to know that a file or path is *indeed* a directory. The *Directory* class can help us with the aforementioned task. In order to begin operating on a directory, you must initialize the *Directory* class with the *String* path of the directory,

→ *let dir = new Directory(“/path/to/directory”)*

In order to obtain all of the files within the directory—which will not include “.” or “..” files or any special paths—you have two options:

- You may iterate over the *Directory* instance, *dir*, as it is an iterable object:
  - → *foreach file in dir*
- You may return a *List* of files using the *files*() method within the class.

Here are the methods and members that this class implements:

Members:	Description:
files() -> String[]	Returns a *List* of *String*s (*String[]*) of the files in that directory.
filesRecursive() -> String[]	Searches through the directory recursively—it ventures into other directories—and returns a list of all of the files as a *String[]*.
exists() -> Bool	Does the path exist as a directory?
make() -> void	Make the directory.
make(mode: Int) -> void	Make the directory, but with an octal UNIX mode.
delete() -> void	**Recursively** delete the directory and **all** of its file contents—warning!

System

The *System* class provides a way to seamlessly access information about the operating system that the Slow Language Interpreter is running on. It will contain static variables and methods that will help determine the best course of action—no matter which operating system Slow is running on. Here is a list of all methods and variables that it must—statically—implement for it to be compliant:

All methods are static and constant.

Member:	Description:
osName: String	The name of the operating system (e.g., Windows, Linux, or Mac OS).
osDistribution: String	The flavor or edition of the operating system (e.g., 10 (for Windows), Debian (for Linux), or Big Sur (for Mac OS).
osVersion: String	The version number for the OS.
shell(x) -> String	Execute shell command *x*, returning the output of the command.
getenv(x) -> String	Get environment variable *x*.
setenv(x, y) -> void	Set environment variable *x* to *y*.
home: String	Home directory
username: String	Username of who is running the process.
cwd() -> String	Get current working directory.
chdir(dir: *Path*) -> void	Change current working directory—*cd*.
rmdir(dir: *Path*) -> void	Recursively remove a directory—permanently.
remove(file: *Path*) → void	Remove a file or an empty directory.
move(source: *Path*, destination: *Path*) -> void	Move a file or directory from *source* to *destination*.
rename(old: *Path*, new: *Path*) -> void	Rename a file or directory from *old* to *new*. *This is technically identical to move.*

Math

While operators are often sufficient for doing the mathematics required for most programs, it is not always the case: we may sometimes need more advanced mathematical functions at our disposal so that we may program a larger variety of applications. The *Math* class offers this ability and is important as to be included into the core standard library of the *Slow Programming Language*.

Below will be a list of all methods and constants associated with the *Math* class. All methods below will be **static**, for they do not require an instance of the *Math* class to be made—you would not be able to do much with it, anyway:

All trigonometric functions use radians instead of degrees, as with most other languages.

Method (static):	Description:
abs(x)	Returns the absolute value of *x*.
factorial(castable x: *Int*) -> *Int*	Returns the factorial of *x*—which must be an *Int*.
isClose(a, b, castable maxdiff: Decimal) -> *Bool*	Returns whether *a* and *b* are close together within a certain margin specified by *maxdiff* (which must be a *Decimal*)
e: Decimal	Euler's number
epow(x) -> *Decimal*	Raise e to the power of *x*.
ln(x) -> *Decimal*	Natural logarithm
log(x) -> *Decimal*	Base 10 logarithm
log(x, base) -> *Decimal*	Logarithm with base.
sqrt(x) -> *Decimal*	Square root of *x*
root(n, x) -> *Decimal*	Take the *n*th root of *x*.
pi: Decimal	π
tau: Decimal	2π, or τ
root2: Decimal	Square root of 2
root3: Decimal	Square root of 3
cos(x) -> *Decimal*	Cosine of *x* where *x* is in radians
sin(x) -> *Decimal*	Sine of *x* where *x* is in radians
tan(x) -> *Decimal*	Tan of *x* where *x* is in radians
acos(x) -> *Decimal*	Arc cosine of *x* returning radians
asin(x) -> *Decimal*	Arc sine of *x* returning radians
atan(x) -> *Decimal*	Arc tan of *x* returning radians

degrees(x: Decimal) -> *Decimal*	Convert *x* which is of radians to degrees.
radians(x: Decimal) -> *Decimal*	Convert *x* which is of degrees to radians.
cosh(x: Decimal) -> *Decimal*	Hyperbolic cosine.
sinh(x: Decimal) -> *Decimal*	Hyperbolic sine.
tanh(x: Decimal) -> *Decimal*	Hyperbolic tan
acosh(x: Decimal) -> *Decimal*	Hyperbolic arc cosine.
asinh(x: Decimal) -> *Decimal*	Hyperbolic arc sine.
atanh(x: Decimal) -> *Decimal*	Hyperbolic arc tan.

Response

To be brief, a *Response* object is useful for representing data that may come through any means possible within the Slow Programming Language. A *Response* object is very simple, consisting of functions to return the *Bytes* and *String* representation of the response—which is useful if we want to return a response with the option of obtaining the *Bytes* or *String* representation. For *Response* to be efficient, it will be initialized with either a *Bytes* or a *String* such that either initial value will be readily available through their respective functions, where the opposing representation will be converted with the function—saving resources.

The following methods and members will be available:

Members:	Description:
_string: *String*	The *String* representation of the data, if initially provided. Value should be *null* if it was not set via a constructor.
_bytes: *Bytes*	The *Bytes* representation of the data, if initially provided with the constructor. Value should be *null* if it was not set with a constructor.
string() -> *String*	Return either the internal *String* as provided by the constructor, or convert the internal *Bytes* object to a *String*. • *Is the internal String null*?
bytes() -> *Bytes*	Return either the internal *Bytes* object as provided by the constructor, or convert the internal *String* object to a *Bytes* object. • *Is the internal Bytes null?*
response: *Int*	What was the response code, *if applicable*?

Additionally, the *Response* object may be casted to and from the following types, which will effectively model the conversions found in the member above:

- *Response → String*
  - *(String) response*
- *Response → Bytes*
  - *(Bytes) response*

Therefore, take it as an implication to suggest that the castings should be overloaded via special functions within the *Response* object.

URL

A *URL* is an object representing a URL for the purposes of acting on it or passing it to another class or function which may explicitly need a URL and know that it is a valid URL. The *URL* object follows the URL standard as found abundantly. A *URL* object may be initialized with *scheme*:*location*, like so:

→ https://website.com
→ example.org
- Assumed to be of an HTTPS scheme, if a scheme is not present.

→ http://192.168.0.1:8080
- An IP address with or without a port is also valid

→ ftp://ftpserver.org
- The *URL* object does not have the ability to interact with any other protocols than HTTP or HTTPS.

→ file://path/to/file
- Can be casted to a *Path*.

The URL object will incorporate the following methods in its definition:

Members:	Description:
file: *Bool*	Is the URL representing a local file? • If so, then a casting from a *URL* to a *File* is possible.
scheme: *String*	The scheme of the URL
domain: *String*	The domain enclosed in the URL, without the *port* if it is provided.
resolve() -> *String*	If there is a domain present within the URL, return the IP address of that domain if available. If it cannot be resolved If the *URL* object is itself an IP address, return itself.
port: *Int*	The port of the domain • If one is explicitly provided, then *port* will be set to that • If there are none explicitly provided, then *port* will be set to the default port of the recognized schemes: ◦ 80 ◦ 443
get() -> *Response*	Return the response from making a GET request to the URL as a *Response*, allowing it to be received as a *String* or *Bytes*.

	• This will connect to the URL. • This method only works when the URL is of HTTP or HTTPS.
post(data: *Table = null*, json: *Table = null*, files: *Table = null*) -> *Response*	Make a POST request to the URL, providing multipart form data or JSON data content type, along with files—*data* and *json* are mutually exclusive, being provided by keyword arguments. • This will connect to the URL. • This method only works when the URL is of HTTP or HTTPS • If the mutual exclusion is violated, an *ArgumentException* will be promptly thrown.

When specifying files in the *files Table*, you should provide the filename corresponding to a *File* object open in binary reading, like so:

url.post(files = {“filename”: new File(“/path/to/file”, ReadBinary)})

Control Flow

Parentheses, where noted, are required by the Slow Programming Language and will result in an error if they are not present. Indentation is recommended, albeit not required.

while (condition)

    …

end

- Repeats the contents within the block while *condition* evaluates to a *true* condition.
- Any action taken inside *condition* is done within the variable scope of the while block.

for (init; condition; post)

    …

end

- Before executing, executes the expression contained within *init* within the variable scope of the *for* block.
- It will repeat the code inside of the *for* block while *condition* evaluates to *true*.
- After each iteration, the expression inside *post* will be executed within the variable scope.
- All actions taken within each section of the for-loop are done within its scope.

if (condition)

    …

end

- If *condition* evaluates to *true*, the code inside of *if* will be executed.
- The condition taken inside of the *if* statement is **NOT** executed under its scope, instead executing within the scope above it.

else if (condition)

    …

end

- else if must be used in conjunction with a previous if-block, lest *SyntaxException* is returned.
- If the previous if-block did not evaluate to true, evaluate new conditional and execute the code inside of the block; additionally, subsequent else if and else blocks will not execute.
- The condition taken inside of the else if statement is **NOT** executed under its scope, instead executing within the scope above it.

else
    …
end

- else must be used in conjunction with a previous if-block, at least—or *SyntaxException*.
- If none of the conditions evaluated to true, execute the code inside of the else block.

foreach (item in iterable)
    …
end

- Loops through each item in an iterable object and exposes it as item within the local block variable space.
- If you wish to obtain the index of the item compared to iterable, use the *indexof* operator. indexof may return null if the index of the iterable object doesn't make sense—as in the case of reading from stdin.

switch (value)
    case value1
        …
    end

    case value2
        …
    end

    default
        …
    end
end

- Using the value of a variable, preform a particular function given that it equals a value that is handled within the *switch* body.
- Unlike other languages, *break* is not required.
- When none of the cases are matched, the *default* block shall be executed.
- switch is most effective when used with enumerations.

with (fileStream as x)
    x.write(…)
end

- Automatically open and close the file within the block. This is useful for scoping file operations as well as properly managing them without error. *with* will automatically call *open()* and *close()* on the file streams.

continue – Skip past this iteration, proceed to the next one (if applicable)
break – Stop this loop.

- If *continue* or *break* are not used within a loop, a *SyntaxException* is thrown.

Looking for goto? Get better at control flow! Slow does not support goto for the same likely reason that Python does not: we do not store every single instruction in memory, so we cannot even jump back to other lines of code, even if we wanted to!

space
    …
end

- Provides an area where it is scoped without any additional qualities. Consider this to be equivalent to a *{ … }* body in a C-like language without any additional keyword to support it.

python
    …
end

- Executes lines of Python.
- **Ideally**: identation relative to the starting of the *python* block shall be preserved, considering how Python requires proper indentation, as the Slow interpreter will strip whitespace (including identation)
    - Considering the simplicity of implementing the standard library—we will be piggy-backing off of Python's—you should not need complicated code such that it would require an indentation.

Exceptions

Exceptions within the Slow Programming Language are handled similarly to other languages. An exception is a class that is or inherits from the *Exception* class. If a class is thrown while not being a descendent of the *Exception* class, an uncatchable error will be thrown. An Exception may be thrown using the *throw* keyword, followed by a new instance of the aforementioned classes, which may contain a *String* for the constructor detailing the message.

An exception may be caught and handled with a *try … catch … finally* block. In the Slow Programming Language, the catch does not allow for the setting of the variable for the caught exception; the exception instance is instead represented by *~error*. You may catch multiple exceptions by providing multiple *catch* blocks for handling each type of exception individually. You may also group certain types of exceptions by separating them by commas within the *catch* block. For example:

```
try
        throw new Exception()
end
catch (Exception, Exception2, etc)
        print(~error)
end
catch (AnotherException, AnotherException2)
        print(~error) // Handle another exception separately
end
finally
        cleanUp()
end
```

The *Exception* base class will have the following members:

Members:	Description:
line: Int	Line number of where the exception occurred.
file: String	The file in which the exception occurred.
name: String	The name of the exception—children of *Exception* should override this.
message: String	The message of the exception—what happened?
critical: Bool	Is it critical enough to stop the program or be caught? By default, this is *true*. The *Warning* class will have it set to *false*.

The following children of the *Exception* class will be provided in the standard library, to provide generic error handling:

Exception:	Description:
SyntaxException	The interpreter found a syntax error.
ImmutableException	Setting the value of a constant variable.
TypeException	A type-mismatch occurred; the variable does not accept that type.
NullPointerException	Trying to access an instance that is set to *null*.
VoidException	A function's returning of *void* cannot be used—for anything!
IndexException	The provided index does not exist—in an array or a table!
NameException	The name of whatever you are requesting does not exist. • Includes *name: String*, where it includes the whatever you are trying to reference.
RedefinitionException	You may not make another definition of that variable, function, or object. • Includes *name: String*, where it includes what has been redefined.
OperatorException	The class does not support that operator—it has not been implemented by overloading.
CastException	The class supports no method for casting to that particular type *T* **or** the variable does not have the *castable* modifier associated with it.
PathNotFoundException	The path does not exist. • *path: String*, representing the *String* path passed to initialize the *Path* object returning this error.
PathExistsException	The path exists already. • *path*: String, representing the *String* path with the same origination qualities as above, to be brief!
IsADirectoryException	The file is actually a directory: cannot operate on it. • *path: String*, representing the path.
NotADirectoryException	The file is not a directory: cannot operate on it. • *path: String*, representing the path.

PermissionException	You lack the permissions to operate on this file. • *path: String*, representing the path…
MathException	An error relating to mathematical operations has arisen—did you divide by zero?
NotSupportedOnOSException	Self-explanatory.
ArgumentException	The function does not support that argument or will not accept it.
KeyException	The key was not found in that *Table* or *List*.
TerminationException	The process has been terminated.
OSException	General operating system error (e.g., *disk full* or *file not found*). • This object will include *errno: Int* and *strerror: String* including the specific error met along with its English interpretation of it.
RecursionException	The maximum allowed recursion has been met
BrokenPipeException	The pipe or socket does not exist anymore, but you tried to access it.
ConnectionAbortedException	The socket connection was aborted or reset by the peer.
ConnectionRefusedException	The socket connection was refused by the server.
URLException	The URL is incorrect.
ResolutionException	The resolution of that URL's domain to an IP address encountered an error.
Warning	A non-fatal error, which will be printed but will not cause termination of the entire program. If you wish to make a warning, please inherit from this class.
DeprecationWarning	If a certain method becomes deprecated, throw this warning.
BadPracticeWarning	If something is a bad practice, throw this warning.

Use & Using

Due to the fact that importing and including other libraries or sections of code from another file is an integral aspect of a programming language, the Slow Programming Language has implemented keywords which serve those purposes: *use* and *using*. They are similar words, because of the fact that they have similar features—they import files.

Their subtle difference, however, is that while *using* can be used to import any file—with the current working directory set to the directory of the file making the import, *unlike* PHP—only *use* can be used to import a file—assuming the extension is a .slow—from the include/import path of the Slow interpreter—something that is not within the purview of this document, for this document does not discuss the specifications of the Slow interpreter which will implement most of the things detailed throughout this document.

For example:

→ *use a_library_in_include_path*

- As implied, *use* only searches for files with the .*slow* extension, and it does not require an extension when using it as a keyword.

→ *using my_file.slow*

- *using* will directly include the file, where the current working directory will be set to that which *using* has accessed, such that future calls to *using* from those files will be relative to their directory—again, *unlike* PHP!

The Slow Interpreter must keep track of the files that it has imported, lest a multiple definition error will be encountered at some point. The programer shall not have to worry about include guards or anything of the kind; they will with cyclic imports, though!

Namespace

It is useful to partition code such that it is not all combined with each other at once. A *namespace* can achieve this, working similarly to the *namespace* feature in C++ or the *package* feature seen in Java. Despite its name, *namespace* in the Slow Programming Language works most similarly to the *package* keyword in Java, not being a block of code and requiring only one line, for one file. For instance:

→ *namespace std*

In order to access elements from a non-imported namespace, you must use the dot access operator, as you would in most other languages. For instance, a *hello_world()* method inside of the namespace *std*:

→ *std.hello_world()*

It may be of use to—if you are consistently using that aforementioned function, for instance—import it on a global scale. You may use the *import* keyword, like so:

```
use std
import std.hello_world

hello_world()
```

- Note: you may import everything within a namespace—not recursively though—using the asterisk after the dot: *import std.**

In spite of the fact that in our previous examples, we showcased the example namespace as *std*, making the implication that the standard library will be under a namespace: this is not true. In the Slow Programming Language, the core standard library defined in the previous pages of this specification is not under a namespace, but in the global space.

It is claimed that this is a bad practice, where *using namespace std;* is considered to be an unholy practice amongst those who deal with C++. This is a false equivalency, however, due to the fact that the naming of functions and objects within the Slow Programming Language's standard library is more complicated than those found within C++'s standard library, because of the full length words used along with camel case. It is, therefore, true that name collisions will virtually never occur under these naming conventions.

It is obligatory, however, that extensions to the standard library that are not core features of the Slow Programming Language (e.g., JSON, REGEX, etc) be under their own namespace, in a similar fashion to how Python accomplishes that. A core function or object of the standard library is any function or object described in this specification alone.

Commenting

Commenting is such an integral aspect of the programming process that the Slow Programming Language wishes to have all forms of commenting which are efficient in their specific use cases: one-line comments and multi-line comments.

In order to make a one-line comment, you have some choices:

- Using a #: *# This is a comment*. This is useful for shebangs, if using the Slow Programming Language for scripting.
- Using //: *// This is also a comment*. This is preferred in this specification, as evidenced by the comments which have already appeared throughout this document.

Conversely, one may wish to make a multi-line comment, which can be done by using a /* … */ block, akin to comments in C-style programming languages. For example:

/ This*
is
a multi-line
comment.
**/*

- Note: the /* … */ block does not strictly have to be on newlines, as the following information will inform you.

Comments may also appear on lines of instructions which shall actually be executed. Let us illustriate this with two styles of commenting:

- *doStuff() // This comment is valid*
- *doStuff() /* This comment is also valid. */*

.
.
.
.
.
.
.
.
.
.
.
.
.

There is nothing else to say. Damn, I really tried hard to fill this entire page up, as you can tell!

Embedded Slow & CGI (if applicable!)

The Slow Programming Language supports embedding it into HTML or into other programming languages to serve as a preprocessor of some sorts, which may be useful for CGI applications or anything of the like; Slow would be similar to PHP in this regard. In fact, the method which the Slow Programming Language uses in order to embed itself into other programming languages and scripts is quite similar: wherever Slow's output is to be used within a file, it shall be encased within a block of *<?slow … ?>*. For example:

```
<body>
	<?slow
		print((String) 2 * 2) // Explicit cast not required, but here for showcasing purposes.
	?>
</body>
```

Shall format that aforementioned block into something returning this:

```
<body>
	4
</body>
```

- Anything that is not within the *<?slow … ?>* block is merely echoed back, as you can see.

By default, this feature shall be disabled and Slow shall not require a *<?slow … ?>* block in order to execute. The Slow interpreter shall check for an environment variable—additionally a command line parameter—in order to activate this feature, which shall be trivial to implement.

CGI

CGI scripts of Slow may be implemented by the interpreter, where the interpreter will set the variables *~POST, ~GET, ~FILES, ~SERVER, ~SESSION, ~COOKIES,* and *~REQUEST* which will serve as variables to interface with the CGI protocol. The interpreter shall be in charge of everything else and this document does not have much bearing on how it wishes to implement the CGI protocol.

See the CGI Protocol for reference. Due to the fact that the CGI protocol is outdated and is not the industry-standard, Slow cannot be taken seriously—which is okay, due to the fact that it only exists as a proof of concept.