Table of Contents

Introduction

FatScript logo

Hello World

```
_ <- fat.std
console.log('Hello World')
```

Quick Start

Jump straight into the docs:

- General overview
- Language syntax
- Standard libraries

Running your code

You can run FatScript using either the `fry` interpreter or the web playground.

Fry Interpreter

For local execution, use the `fry` interpreter. For details on its installation and usage, refer to the setup section.

Web Playground (beta)

For quick and convenient testing, run your code directly in the FatScript Playground. The playground features a REPL and an intuitive interface that allows you to load scripts from a file, facilitating swift experimentation.

PDF Download

- FatScript v2.6.0 (current)
- FatScript v1.3.5 (legacy)

Tutorials

Dive into our immersive tutorials, behind-the-scenes insights, and surrounding topics in the FatScript YouTube channel.

Donations

Did you find FatScript useful and would like to say thanks?

Buy me a coffee

License

fatscript.org

Published on Mon Jun 10 2024 00:35:04 GMT+0100 (British Summer Time)

General overview

FatScript is a lightweight, interpreted programming language designed for building console-based applications. It emphasizes simplicity, ease of use, and functional programming concepts.

Free and open-source

`fatscript/fry` is an open-source project that encourages knowledge sharing and collaboration. We welcome developers to contribute to the project and help us improve it over time.

Key Concepts

- Automatic memory management through garbage collection (GC)
- Symbolic character combinations for a minimalistic syntax
- REPL (Read-Eval-Print Loop) for quick expression testing
- Support for type system, inheritance, and sub-typing via aliases
- Support for immutable programming and passable methods (as values)
- Keep it simple and intuitive, whenever possible

Contents of this section

- Setup: how to install the FatScript interpreter
- Options: how to customize the runtime
- Bundling: how to pack a FatScript application
- Tooling: overview of a few extra tools and resources

Limitations and challenges

While FatScript is designed to be simple and intuitive, it is still a relatively new language and may not be suitable for all use cases. For example, it may underperform compared to more mature programming languages when dealing with complex workloads or high-performance computing tasks.

Setup

To start "frying" your fat code, you'll need an interpreter for the FatScript programming language.

fry, The FatScript Interpreter

fry is a free interpreter and runtime environment for FatScript. You can install it on your machine using the following instructions.

Installation

`fry` is designed for GNU/Linux, but it might also work on other operating systems.

For Arch-based distributions, install via fatscript-fry AUR package.

For other distributions, try the auto-install script:

```
curl -sSL https://gitlab.com/fatscript/fry/raw/main/get_fry.sh -o get_fry.sh;
bash get_fry.sh || sudo bash get_fry.sh
```

Or, to install `fry` manually:

- Clone the repository:

```
git clone --recursive https://gitlab.com/fatscript/fry.git
```

- Then, run the installation script:

```
cd fry
./install.sh
```

> the manual installation may copy the `fry` binary to the $HOME/.local/bin folder, alternatively use `sudo` to install it to /usr/local/bin/

- Verify that `fry` is installed by running:

```
fry --version
```

Dependencies

If the installation fails, you may be missing some dependencies. `fry` requires `git`, `gcc` and `libcurl` to build. For example, to install these dependencies on Debian/Ubuntu, run:

```
apt update
apt install git gcc libcurl4-openssl-dev
```

Back-end for text input

`linenoise` is a lightweight dependency and an alternative to `readline`, maintained as a submodule. If it was not included during the initial `git clone` operation, you can rectify this with the following commands:

```
git submodule init
git submodule update
```

If you prefer to link against `readline`, just ensure it is installed by running:

```
apt install libreadline-dev
```

OS Support

`fry` is primarily designed for GNU/Linux, but it's also accessible on other operating systems:

Android

If you're on Android, you can install `fry` via Termux. Just install the required dependencies like so:

```
pkg install git clang
```

Then you can follow the standard installation instructions for `fry`.

ChromeOS

If you're using ChromeOS, you can enable Linux support by following the instructions here.

MacOS

If you're using MacOS, you'll need to have Command Line Tools installed.

iOS

If you're using iOS, you may use `fry` via iSH. First, install the required dependencies:

```
apk add bash gcc libc-dev curl-dev
```

Then, according to this thread, configure `git` to work properly, like so:

```
wget https://dl-cdn.alpinelinux.org/alpine/v3.11/main/x86/git-2.24.4-r0.apk
apk add ./git-2.24.4-r0.apk
git config --global pack.threads "1"
```

Windows

If you're using Windows, you can use `fry` via Windows Subsystem for Linux (WSL).

Docker image

`fry` is also available as a docker image:

```
docker run --rm -it fatscript/fry
```

To execute a FatScript file with docker, use the following command:

```
docker run --rm -it -v ~/project:/app fatscript/fry prog.fat
```

> replace ~/project with the path to your FatScript file

Troubleshooting

If you encounter any issues or bugs while using `fry`, please open an issue.

Options

With this breakdown of the available modes and parameters you will find out that `fry` has got several spices under the hood for you to better season your runtime.

Command-line arguments

The CLI front-end offers some modes of operation:

- `fry [OPTIONS]` read-eval-print-loop (REPL)
- `fry [OPTIONS] FILE [ARGS]` execute a FatScript file
- `fry [OPTIONS] -b/-o OUT IN` create a bundle
- `fry [OPTIONS] -f FILE...` format FatScript source files

Here are the available option parameters:

- `-a, --ast` print abstract syntax tree only
- `-b, --bundle` save bundle to outfile (implies -p)
- `-c, --clock` time and benchmark logs (toggle)
- `-d, --debug` enable debug logs (implies -c)
- `-e, --error` continue on error (toggle)
- `-f, --format` indent FatScript source files
- `-h, --help` show this help and exit
- `-i, --interactive` enable REPL with file execution
- `-j, --jail` restrict FS, network and sys calls
- `-k, --stack` # set stack depth (frame count)
- `-m, --meta` show info about this build
- `-n, --nodes` # set memory limit (node count)
- `-o, --obfuscate` encode bundle (implies -b)
- `-p, --probe` perform static analysis (dry run)
- `-s, --save` store REPL session to repl.fat
- `-v, --version` show version number and exit
- `-w, --warranty` show disclaimer and exit
- `-z, --minify` minify source (implies -p)

the `-e` option is auto-enabled with REPL and probe modes

combining `-p` with `-f` sends formatted result to stdout

Memory management

`fry` manages memory automatically without pre-reservation. You can limit memory usage by specifying the number of nodes with CLI options:

- `-n <count>` for an exact node count
- `-n <count>k` for kilonodes, count * 1000
- `-n <count>m` for meganodes, count * 1000000

For example, `fry -n 5k mySweetProgram.fat` restricts the app to 5000 nodes.

The garbage collector (GC) runs automatically when there are 256 nodes left before the final memory limit is reached (GC premonition). You can also invoke the GC at any time by calling the `runGC` method of SDK lib from the main thread.

Bytes estimate (x64)

Each node on a 64-bit platform uses approximately ~200 bytes. The actual node size depends on the data it holds. For example, the default limit is 10 million nodes, your program can use up to 2 GB of RAM when reaching the default limit.

Use the `-c` or `--clock` option to print the execution stats to have a better understanding of how your program is behaving in practice.

Runtime verification

There are two embedded commands for checking memory usage at runtime:

- `$nodesUsage` currently allocated nodes (O(1))
- `$bytesUsage` currently allocated bytes (O(n))

checking the currently allocated bytes is an expensive operation as it needs to traverse all nodes to check the actual size of each one

Stack size

The maximum stack depth is defined in `parameters.h`, however you may be able to customize the stack size up to a certain point using CLI options:

- `-k <count>` for an exact frame count
- `-k <count>k` for kibiframes, count * 1024

Run commands file

On bootstrap, `fry` looks for a `.fryrc` file on the same path of the program file and, if not present, also on the current working directory. If found, it is executed as a "precook" phase to set up the environment for the program execution.

Memory management with .fryrc

You can use the `.fryrc` file to define the memory limit for your project without needing to specify it as a CLI argument. To do this, you can use the `setMem` method provided by the SDK lib, like this:

```
_ <- fat.system
setMem(64000)  # sets 64k nodes as memory limit
```

Bootstrap details

CLI options are applied first, except for the memory limit. During the precook phase, `fry` uses the default limit of 10 million nodes, regardless of the CLI option. If you define a memory limit in the `.fryrc` file, that limit takes effect from that point on and overrides the CLI option for the whole execution. If the `.fryrc` file does not set a memory limit, the CLI option takes effect after the precook phase.

The precook scope is invisible by default. After the `.fryrc` file is executed, a fresh scope is provided for your program, which allows you to test your code with a very low limit of nodes when using a `.fryrc` file without affecting the node count. This also prevents the `.fryrc` namespace from clashing with your program's global scope. However, if you want to keep the entries declared in `.fryrc` in the global scope for configuration purposes, you can call the embedded command `$keepDotFry` somewhere in the `.fryrc` file.

Another possible use, other than setting up memory limit, is to pre-load common imports, for example the standard types:

```
$keepDotFry
_ <- fat.type._
```

Sandbox mode

Use the `-j` or `--jail` option to inhibit the following embedded commands:

- `write`, `remove`, and `mkDir` - These commands modify the file system.
- `request` - This command is used for making outbound HTTP requests.
- `shell`, `capture`, `fork`, and `kill` - These commands are involved in starting or stopping arbitrary processes.

See also

- Embedded commands
- SDK library

Bundling

Fry offers an integrated bundling tool for FatScript code.

Usage

To bundle your project into a single file starting from the entry point, execute:

```
fry -b sweet mySweetProject.fat
```

This process consolidates all imports (except literal paths) and trims unnecessary spaces, enhancing load times:

- Adds a shebang to bundled code
- Receives the execute attribute for file mode

Subsequently, you can run your program:

```
./sweet
```

> the bundling will replace any `$break` statements (debugger breakpoint) with `()`

Obfuscating

For optional obfuscation, use `-o`:

```
fry -o sweet mySweetProject.fat  # creates the obfuscated bundle
./sweet                          # executes your program as usual
```

> When distributing via public hosts, consider setting a custom key with a local `.fryrc`. Only the client should be privy to this key to safeguard the source.

Obfuscation leverages enigma algorithm for encryption, ensuring swift decoding. For optimal load times, prefer `-b` if obfuscation isn't essential.

Caveats

Imports are deduplicated and inlined based on their order of first appearance. As a result, the sequence in which you import your files could play a role in the final bundled output. Though these considerations are usually inconsequential for small projects, bundling larger projects may require additional organization. Always validate your bundled output.

Tooling

Here are a few hints that can enhance your coding experience with FatScript.

Static analysis

Use the probe mode to check the syntax and receive hints about your code:

```
fry -p mySweetProgram.fat
```

Debugger

A breakpoint, indicated by the command `$break`, serves as a debug tool by temporarily halting the program execution at a designated location and loading the built-in debugging console. It provides an interactive environment for examining the current state of the program by inspecting values in scope, evaluating expressions, and tracing program flow.

To activate breakpoints, it is necessary to run the program with interactive mode enabled:

```
fry -i mySweetProgram.fat
```

In FatScript, `$break` returns `null`, which can alter a return value if placed at the end of a block, due to the auto-return feature. Be cautious with `$break` placement to avoid unintended effects on program functionality.

Source code formatting

Built-in support

You can apply auto-indentation to your sources using the following command:

```
fry -f mySweetProgram.fat
```

Visual Studio Code Extension

To add code formatter support to VS Code, you can install the fatscript-formatter extension. Launch VS Code Quick Open (Ctrl+P), paste the following command, and press enter:

```
ext install aprates.fatscript-formatter
```

> `fry` needs to be installed on your system for this extension to work

Syntax highlighting

Visual Studio Code Extension

To add FatScript syntax highlighting to VS Code, you can install the fatscript-syntax extension. Launch VS Code Quick Open (Ctrl+P), paste the following command, and press enter:

```
ext install aprates.fatscript-syntax
```

You can also find and install these extensions from the VS Code Extension Marketplace.

Vim and Neovim Plugin

To install FatScript's syntax highlighting for Vim and Neovim, check out the vim-syntax plugin.

For Neovim users, add the respective line to your configuration:

Using packer.nvim:

```
use { 'https://gitlab.com/fatscript/vim-syntax', as = 'fatscript' }
```

Using lazy.nvim:

```
{ 'https://gitlab.com/fatscript/vim-syntax', name = 'fatscript' }
```

Nano Syntax File

To install FatScript's syntax highlighting for `nano`, follow these steps:

1. Download the `fat.nanorc` file from here.
2. Copy the `fat.nanorc` file to the `nano` system directory:

```
sudo cp fat.nanorc /usr/share/nano/
```

If the syntax highlighting does not get automatically enabled, you may need to explicitly enable it in your `.nanorc` file. Refer to the instructions in the Arch Linux Wiki for more information.

After installing the syntax highlighting, you can also use the code formatter in `nano` with the following shortcut sequence:

- Ctrl+T Execute; and then...
- Ctrl+O Formatter

Other tips

Console file navigation

To navigate your project folders from the terminal, you can try using a console file manager such as ranger, paired with `nano`, `vim` or `nvim`. Set it as the default editor for `ranger` by adding the following line to your `~/.bashrc` file:

```
export EDITOR="nano"
```

Syntax

In the following pages, you will find information on the central aspects of writing FatScript code, using both the basic language features as well as the advanced type system and standard libraries features.

Topics covered

- Formatting: how to format FatScript code properly
- Imports: how to import libraries into your code
- Entries: understanding the concept of entries and scopes
- Types: a guide to FatScript type system
- Flow control: controlling the program execution with conditionals
- Loops: making use of ranges, map-over and while loops

Formatting

In FatScript, whitespace and indentation are irrelevant, yet they are very welcome to make the code more readable and easier to understand.

Whitespace

- A newline character (`\n`) indicates the end of an expression, except when:
  - the last token on the line is an operator
  - the first token of the next line is a non-unary operator
  - using parentheses to group expressions
- Expressions can be on the same line if separated by comma (`,`) or semicolon (`;`)

Comments

Comments start with #, and are terminated by a newline:

```
a = 5  # this is a comment
```

Note

FatScript does not support multiline comments at the moment. Additionally, text literals may end up as a valid return value if left as the last standing line, due to the auto-return feature. Therefore, it is recommended to stick to the single line comment format.

See also

- Source auto-formatter

Imports

Let's unravel the art of importing files and libraries in FatScript! Why? Well, because in this language you can import whenever your heart desires, simply by using a left arrow `<-`.

Dot syntax

To use imports with dot syntax, project files and folders should neither start with a digit nor contain symbols.

> you can force any path you like by using literal paths

Named import

To import files, use the `.fat` extension for filenames (or no extension at all), but omit the extension in the import statement. Here's an example:

```
ref <- filename
```

> if both `x` and `x.fat` files exist, the latter takes precedence

For importing files from folders:

```
ref1 <- folder.filename
ref2 <- folder.subfolder.filename
```

To import all files from a folder, leverage the dot-underscore syntax:

```
lib <- folder._
```

Please note: only files immediately inside the folder are included using the above syntax. To include files from subfolders, explicitly mention them. Additionally, a "_.fat" file (or "_" file) inside a folder can override the dot-underscore import behavior.

Element access

Once imported, access elements using dot syntax:

```
ref1.element1
```

Element extraction

To extract specific elements from a named import or to avoid prepending the module name every time (e.g., `lib.foo`), employ destructuring assignment:

```
{ foo, bar } = lib
```

Visibility

Named imports are resolved at the global scope, irrespective of where they are declared. This means even if you declare a named import inside a function or a local scope, it will be globally accessible.

Local import

To import within the current scope, use:

```
_ <- filename
```

Local imports, unlike named imports, dump the file content directly into the current scope. Thus, an imported method can be invoked as `baz(arg)` rather than `ref.baz(arg)`.

While local imports are best suited for importing types into the global scope, they should be used with caution when importing library content. Overusing local imports can lead to namespace pollution, which can make it more challenging to follow the code, because it becomes less apparent where the methods come from.

Selective local import

You also can discard elements from a local import by using destructuring assignment:

```
{ foo } = { _ <- lib }
```

the point is to avoid namespace pollution, as all the contents will be processed

Literal paths

With literal paths, you may use any filename or extension. However, note that those imports are not evaluated during bundling, but at runtime. Here's an example:

```
ref <- '_folder/2nd-source.other'
```

You can also use smart texts as literal paths:

```
base = 'folder'
file = 'source.xyz'
ref <- '{base}/{file}'
```

Since FatScript alternatively accepts JSON-like syntax you may even load a JSON file directly as an import:

```
json <- 'sample/data.json'
```

however possible, it is more advisable to use file.read and then recode.fromJSON

Keep in mind that literal paths can make your code more complex, and those imports can only be dynamically resolved, so use them sparingly.

Import policy

FatScript utilizes an "import once" strategy with an in-scope flag mechanism, automatically bypassing files that have already been imported.

Imports are generally resource-efficient. However, **local imports within method bodies** should be avoided as they are re-evaluated with each invocation, potentially causing memory retention.

This behavior is not classified as a bug per se, but rather a consequence of design choices in FatScript's garbage collection (GC) system. The GC's optimizations exclude nodes directly derived from source code, allowing them to evade standard mark-and-sweep procedures. As a result, local imports within methods miss out on deduplication, causing their nodes to remain resident until the program's end:

```
myMethod = -> {
  _ <- lib  # potential memory leak
  ...
}
```

Here are some strategies to address this issue:

- Relocate the import statement to the outer scope.
- Opt for a named import as an alternative.
- Reorganize the 'lib' structure to export a method.

Entries

Entries are key-value pairs that exist in the scope where they are declared.

Naming

Entry names (keys) **cannot** start with an uppercase letter, which is the distinction compared to types. Identifiers are case-sensitive, so "frenchfries" and "frenchFries" would be considered different entries.

The recommended convention is to use `camelCase` for entries.

> you may use an arbitrary name as key by using dynamic nomination

Declaration and assignment

In FatScript, you can declare entries by simply assigning a value:

```
isOnline: Boolean = true
age: Number       = 25
name: Text        = 'John'
```

Types can also be inferred from assignment:

```
isOnline = true    # Boolean
age      = 25      # Number
name     = 'John'  # Text
```

Immutable entries

In FatScript, declaring an entry defaults it to being immutable, meaning once assigned, its value cannot be changed. This immutability ensures consistency throughout the program's execution:

```
fruit = 'banana'
fruit = 'apple'  # raises an error because 'fruit' is immutable
```

Exception to the Rule

The immutability in FatScript applies to the binding of the entry, not to the contents of scopes. Even though an entry is immutable, if it contains a scope, the content of that scope can be modified, either by adding new entries or by modifying mutable entries within the scope:

```
s = { a = 1, b = 2 }
s.c = 3  # even though 's' is immutable it accepts the new value of 'c'
s        # now { a = 1, b = 2, c = 3 }
```

This design choice offers flexibility with scope modifications. In contrast, lists enforce stricter immutability, preventing the addition of new entries to immutable lists.

Also note that scopes are always passed by reference. To modify a scope's content without altering its original reference, use the `copy` method from the Scope prototype extension to create a duplicate.

Mutable entries

Yes, you can declare mutable entries, also known as variables. To declare a mutable entry, use the tilde ~ operator:

```
~ fruit = 'banana'
fruit   = 'apple'  # ok
```

Note that even a mutable entry cannot immediately change its type, unless it's erased from the scope. To erase an entry, assign `null` to it, and then redeclare it with a new type. Changing types is discouraged by the syntax and not recommended, but it is possible:

```
~ color = 32    # creates color as a mutable Number entry
color = 'blue'  # raises a TypeError because color is a Number
```

```
color = null    # entry is erased
color = 'blue'  # redefines color with a different type (Text)
```

you have to declare the entry as mutable again using tilde ~ when redefining after erasure if you want the next value to be mutable

Dynamic entries

You can create entries with dynamic names using square brackets `[ ref ]`:

```
ref = 'popCorn'  # text will be the name of the entry

options = { [ ref ] = 'is tasty' }

options.[ref]    # dynamic syntax: yields 'is tasty', with read and write access
options(ref)     # get syntax:     yields 'is tasty', but value is read-only
options.popCorn  # dot syntax:     yields 'is tasty', but has to follow naming rules
```

all dynamic declarations are mutable entries

This feature allows to dynamically define the names inside a scope and create entries with names that otherwise would not be accepted by FatScript.

Dynamic entries can also use numeric references, however the reference is converted into text automatically, e.g.:

```
[ 5 ] = 'text stored in entry 5'
self.['5']                      # yields 'text stored in entry 5'
self.[5]                        # yields 'text stored in entry 5'
```

in a different context, not followed by assignment = or preceded by dot notation `.`, dynamic syntax will be interpreted as a list declaration

Special entries (DEPRECATED)

Entries with names starting with underscore _ are completely free and dynamic, they don't require tilde ~ and can also change type without the need of erasure, like variables in JavaScript or Python.

To allow for more aggressive optimization in the interpreter, "special entries" will be deprecated starting from version `3.x.x`. Instead, they will be treated as immutable, unless declared with ~, indicating that they are mutable, but won't be able to change type without need of erasure. That is, there will be no special treatment for entries with names that start with an underscore. That is, there will be no special treatment for entries with names that start with an underscore.

Destructuring assignment

You can copy values of a scope into another scope like so:

```
_ <- fat.math
distance = (position: Scope/Number): Number -> {
  { x, y } = position    # destructuring assignment into method scope
  sqrt(x ** 2 + y ** 2)  # calculates distance between origin and (x, y)
}
distance({ x = 3, y = 5 })  # 5.83095189485
```

The same syntax works similarly for lists:

```
distance = (position: List/Number): Number -> {
  { x, y } = position    # extracts first and second items to 'x' and 'y'
  sqrt(x ** 2 + y ** 2)
}
distance([ 3, 5 ])  # 5.83095189485
```

You can also use destructuring assignment to expose a certain method or property from a named import:

```
console <- fat.console
{ log } = console
log('Hello World')
```

using this syntax with imports, you can choose to bring to the current scope only the elements of the library that you are interested in using, thus avoiding polluting the namespace with names that would otherwise have no use or could clash with those of your own writing

JSON-like syntax

FatScript also supports JSON-like syntax for declaring entries:

```
"nothing": null,                # Void entry - distinct behavior, see bellow
"isOnline": true,               # Boolean entry
"age": 25,                      # Number entry
"name": "John",                 # Text entry
"tags": [ "a", "b" ],           # List entry
"options": { "prop": "other" }  # Scope entry
```

However it might appear that declaring "nothing" creates a "nothing" value of `null`, it's important to note that the "resulting entry" doesn't actually exist in the scope. When you try to access that "nothing", FatScript does return `null`, but if you attempt to map over the scope, the name of that entry will be missing since it was never truly created.

It's important to note that JSON-like declarations always create immutable entries, so you can't prepend them with the tilde ~ character to make them mutable.

Types

Types are used in FatScript to combine data and behavior, acting as templates for creating new replicas.

Naming

Type names are case-sensitive and must start with an uppercase letter.

The recommended convention for type identifiers is `PascalCase`.

Native Types

FatScript provides several native types:

- Any - anything
- Void - nothing
- Boolean - primitive
- Number - primitive
- HugeInt - primitive
- Text - primitive
- Method - function or lambda
- List - like array or stack
- Scope - like object or dictionary
- Error - yes, for errors
- Chunk - binary data

However, you need to import the types package to access the prototype members for each type.

Additional Types

FatScript's native types are augmented with a collection of extra types that build upon the core functionalities of its native types. Crafted in pure FatScript, these additional types cater to various advanced programming needs and facilitate common design patterns.

Moreover, you will find domain-specific types embedded within libraries, such as `Worker` in the async library, `FileInfo` in file, `HttpRequest` (among others) in http, `CommandResult` in system etc.

Custom Types

Besides using the types provided by the language or an external library, you may also create your own types, or extend existing ones with new behaviors.

Declaration

To define a custom type in FatScript, you can use a simple assignment statement. The type definition can be wrapped in either parentheses or curly brackets. Both syntaxes are valid and have the same effect. You may also optionally define default values for the type's properties, as shown in the following example:

```
# Type definition using parentheses with default values
Car = (km: Number = 0, color: Text = 'white')
```

Global Uniqueness

FatScript features a singular global meta-space, necessitating unique type names across your entire program and any included libraries. Attempting to define a type that shares a name with an existing type, even if in a different scope, triggers an `AssignError`. However, if the new definition is identical, it will simply be ignored.

To survey the types present in the global meta-space, the command `_<-fat.std; sdk.getTypes;` proves useful. This function enumerates all defined types, and details their definition locations with `source:line:column` markers. This feature helps navigating and understanding the structure of your code and its dependencies.

It is wise to steer clear of names already in use by `fat.std` library types when defining new types.

While FatScript does not impose a strict naming protocol for library development, adopting a conflict-averse naming strategy is recommended. A common practice involves prefixing type names with some unique identifier that reflects your library's name, thereby reducing the likelihood of name clashes.

Usage

To create instances of a custom type, call the type name as if it were a method, optionally passing values for the properties:

```
# Type usage from defaults
car = Car()
# outputs: { km: Number = 0, color: Text = 'white' }

# Type usage defaulting one of the properties
redCar = Car(color = 'red')
# outputs: { km: Number = 0, color: Text = 'red' }

# Type usage, fully qualified
oldCar1 = Car(color = 'blue', km = 38000)
# overrides both values

# Type usage, args using props sequence
oldCar2 = Car(41000, 'green')
# overrides values using type definition order
```

By default, custom types return a scope of their properties. If you define an `apply` method, however, the type can return a different value. For example, here's a custom type `Sum` with an `apply` method that returns the sum of its `a` and `b` properties:

```
Sum = (a: Number, b: Number, apply = -> a + b)
Sum(1, 2)  # output: 3
```

> note that apply methods do have direct access to instance props

In this example, the output base type of `apply` is a number, not a scope. This also means that the original properties of the custom type are lost during instantiation and cannot be accessed again.

Prototype members

Those are special kind of methods, stored inside the type definition:

```
TypeWithProtoMembers = {
  ~ a: Number
  ~ b: Number

  setA = (newA: Number) -> self.a = newA
  setB = (newB: Number) -> self.b = newB
  sum  = (): Number     -> self.a + self.b
}
```

In this example, `setA`, `setB` and `sum` are prototype members. Note that we needed to use `self`, which is a keyword that provides a self reference to the instance (or method) scope, so that we could gain access to the props.

Checking types

If you're unsure about the type of an entry, you can simply check by comparing it with a type name:

```
place = 'restaurant'
place == Number   # false
place == Text     # true
```

> alternatively, use the `typeOf` method from the SDK library to extract the type name

Anything can be compared with the reserved word `Type` which identifies if it refers to a type:

```
Number == Type  # true
```

`Type` can also be used to specify that a method takes a type parameter:

```
combine = (t: Type, val: Any): Any -> ...
```

Type alias

In FatScript, you can create subtypes by aliasing an existing type. This means that the new type will inherit all of the properties of the base type. Here's an example:

```
_ <- fat.type.Text
Id = Text  # creates an alias
```

Note that type aliases are hierarchical and can be used to classify values while still inheriting the same behavior. However, while the alias is considered equal to the base type, instances of the new type are not considered equal to the base type.

To check if a value is an instance of a type alias or its base type, you can use the less-equal comparison operator `<=`. This allows you to accept any type on the alias chain, down to the base type. Here's an example:

```
Id == Text   # true, as Id is an alias of Text
x = Id(123)  # id: Id = '123'
x == Text    # false, however x is Id it's not Text
x == Id      # true, as expected x is of type Id
x <= Text    # true, as x is of Id which is an alias of Text
```

This feature allows for fine-grained matching on specific types, while still maintaining the flexibility to use different aliases for the same underlying type.

> limitation: it is not possible to create aliases for `Any`, `Type` or `Method`

Type constraints

In FatScript, you can declare type constraints for method parameters. When a method is called, the argument is automatically checked against the type constraint. If the argument is not of the expected type or one of its subtypes, a `TypeError` is raised.

If the type constraint is a base type, any subtype of that type is also accepted as an argument. However, if the type constraint is a subtype, only arguments that match the subtype are accepted. Here's an example:

```
generalist = (x: Text) -> x
restrictive = (x: Id) -> x
```

In this example, the `generalist` method accepts both `Text` and `Id` arguments, because `Id` is a subtype of `Text`. The `restrictive` method only accepts `Id` arguments and not `Text` arguments, because `Id` is a subtype of `Text`, but not the other way around.

It's important to emphasize that custom types are derived from `Scope`. In this context, `Scope` would be the generalist type for, for instance, the custom type `Car`.

Mixin (advanced)

When defining a type, you can add the features of an existing type simply by mentioning it on the type definition. This is called type inclusion or mixin.

For instance, to create a new type `RentalCar` with the properties of `Car` and an additional `price` property, you can write:

```
RentalCar = {
  # Includes
  Car

  # Additional prop
  price: Number
}

RentalCar(50)  # { color: Text = 'white', km: Number = 0, price: Number = 50 }
```

If a property is not defined in the new type, it will inherit the default value from the included type. In the above example, the `color` and `km` properties of `Car` are present in `RentalCar`, with their default values.

Inheriting prototype methods

Suppose we continue from the previous example of type `TypeWithProtoMembers` that has two properties `a` and `b`, and three prototype methods `setA`, `setB` and `sum`. To create a new type `WithMoreMembers` that adds a property `c`, a method `setC` and overrides the `sum` method, you can write:

```
WithMoreMembers = {
  # Includes
  TypeWithProtoMembers

  # Props (instance parameters)
  ~ a: Number
  ~ b: Number
  ~ c: Number

  # Prototype members (methods)
  setC = (newC: Number) -> self.c = newC
  sum  = (): Number     -> self.a + self.b + self.c
}
```

> redeclaring the props allows the new type to also accept arguments at instantiation time, e.g.: `WithMoreMembers(1, 2, 3)` sets `a`, `b` and `c`

When creating a new instance of `WithMoreMembers`, all four prototype methods `setA`, `setB`, `setC` and `sum` will be available.

Note that if there is a redefinition of a property or method in the new type, the new definition takes precedence.

Type casting

In FatScript, the `*` symbol is used for type casting, allowing you to treat one data type as another without altering the underlying data. This capability is especially useful for explicitly specifying the type or for treating values as compatible types, for example:

```
time.format(Epoch * 1688257765448)  # treats the number as a Unix Epoch value
```

Flexible type acceptance

FatScript offers flexibility of type acceptance by implementing a system based on type inclusion. This creates interrelated types that can be interchangeably used within a method or as List items.

When you define a type, it's possible to incorporate one or more additional types within that definition. Take, for example, types `A`, `B`, and `C`. If types `B` and `C` both include type `A` in their definitions, then they are seen as sharing the same set of characteristics derived from `A`. This means `B` and `C` are viewed as sibling types under the umbrella of `A`.

This system enables a method that is designed to accept an object of type `B` to also be capable of accepting an object of type `C`, and vice versa. This is due to the fact that both types `B` and `C` share a common basis in type `A`.

Here's how it looks in code:

```
A = (_)
B = (A, b = true)
C = (A, c = true)

# method1 accepts both B and C because they both include A
method1 = (a: A) -> ...

# method2 accepts C since both B and C include the same set of types
# (making them sibling types)
method2 = (x: B) -> ...

# this logic also applies to List types, as seen with mixedList
mixedList: List/A = [ B(), C() ]
```

> type flexibility is only possible if the data type is based on `Scope`

Caveat

You may have to explicitly check the type, e.g. `x == B` inside the method body if you only want to handle `B`, but not `C` on your method. Or you can create an alias, e.g. `D = A` and use `C = (D, c = true)` as type inclusion to avoid flexible behavior altogether.

Composite types

In FatScript, composite types allow you to define complex data structures composed of simpler types. They are represented using slashes `/` to separate the types within the composite type definition.

Let's go through a few examples and understand how composite types work:

1. `ListOfNumbers = List/Number`, defines a composite type `ListOfNumbers`, which is a list that can only contain numbers.

2. `Matrix = List/List/Number`, defines a composite type `Matrix`, which is a list of lists that can only contain numbers.

3. `MethodReturningListOfNumbers = Method/ListOfNumbers`, defines a composite type `MethodReturningListOfNumbers`, which is a method that returns a `ListOfNumbers`.

4. `NumericScope = Scope/Number`, defines a composite type `NumericScope`, which is a scope whose entries can only be of type number.

See also

- Type package

Any

A virtual type that encompasses all types and no types at the same time.

Default type

`Any` is the inferred type and return type when no type is explicitly annotated in a method. For example:

```
identity = _ -> _
```

is equivalent to:

```
identity = (_: Any): Any -> _
```

Using `Any`, be it implicitly or explicitly, disables type checking for a parameter. The explicit annotation can be a useful in cases where you want to make it clear that you are giving flexibility in the accepted type.

Being too liberal with `Any` can make your code less predictable and harder to maintain. It's generally recommended to be more specific with type annotations whenever possible:

```
# Example of using Any that can lead to issues

console <- fat.console

doubleIt = (arg: Any): Void -> console.log(arg * 2)

doubleIt(2)    # prints: '4'
doubleIt('a')  # yields: Error: unsupported expression > Text <multiply> Number
```

This example shows that although the `Any` type annotation allows flexibility in the parameter type, it can also result in unexpected behavior if an argument of an unexpected type is passed in. By being more specific with the type annotation, such as `Number`, you can make your code more predictable and self-evident.

```
# Example of using a specific type annotation for more predictability

console <- fat.console

doubleIt = (num: Number): Void -> console.log(num * 2)

doubleIt(2)    # prints: '4'
doubleIt('a')  # yields: TypeError: type mismatch > num
```

By using `Number` as the type annotation, the `doubleIt` method is now more specific and only accepts arguments of type `Number`.

Comparisons

The only possible operation with `Any` is comparing to it, but note that `Any` accepts all values indistinctly, so there is no practical use for it:

```
null      == Any  # true
true      == Any  # true
12345     == Any  # true
'abcd'    == Any  # true
[ 1, 2 ]  == Any  # true
{ a = 8 } == Any  # true
```

> comparisons with `Any` can't be used to check for the presence of a value in a scope as even `null` is accepted

Void

When you look into the 'Void', only 'null' can be seen.

Is there anybody out there?

An entry is evaluated to `null` if not defined on current scope.

You can compare with `null` using equality `==` or inequality `!=`, like:

```
a == null  # true, if 'a' is not defined
0 != null  # true, because 0 is a defined value
```

Keep in mind that you can't declare an entry with no value in FatScript.

While you can assign `null` to an entry, it causes different behaviors depending on whether the entry already exists in the scope and whether it's mutable or not:

- If an entry hasn't been declared yet, assigning it `null` has no effect.
- If it already exists and is immutable, assigning `null` raises an error.
- If it already exists and is mutable, assigning `null` removes the entry.

Delete statement

Assigning `null` to a mutable entry is the same as deleting that entry from the scope. If deleted, nothing is remembered about that entry in the scope, not even it's original type.

```
~ m = 4   # mutable number entry
m = null  # deletes m from scope
```

> null "values" are always mutable, as in fact nothing is stored about them, and therefore they are the only kind of "value" that may transition from a mutable state to an immutable state when "reassigned"

Comparisons

You can use `Void` to check against the value of an entry also, like:

```
()    == Void  # true
null  == Void  # true
false == Void  # false
0     == Void  # false
''    == Void  # false
[]    == Void  # false
{}    == Void  # false
```

Note that `Void` only accepts `()` and `null`.

Forms of emptiness

In FatScript, the concept of "emptiness" or the absence of a value can be represented in two ways: using `null` or empty parentheses `()`. They are effectively identical, in terms of behavior in code:

```
null  == null  # true
()    == null  # true
()    == ()    # true
```

Using null

The `null` keyword explicitly denotes the absence of a value. It is commonly used in scenarios where a parameter or return value might not point to any value.

```
method(null, otherParam)
```

```
var = null
```

It can also be used to make a parameter optional, allowing methods to be called with varying numbers of arguments:

```
method = (mandatory: Text, optional: Text = null) -> {
  ...
}
```

`null` can be used explicitly in any context where an absence of value needs to be represented

Using empty parentheses

When used in the context of method returns, `()` can signify that the method does not return any meaningful value.

```
fn = -> {
  doSomething

  ()
}
```

Here, `fn` performs some action and then uses `()` to indicate the absence of a meaningful return value, effectively returning void.

The difference lies in code style, so this is just a suggestion, not a hard rule.

in modern versions of the interpreter, empty parentheses `()` are treated as `null`, ensuring consistent behavior, but, earlier versions required explicitly using `null` to denote the absence of a return value

See also

- Void prototype extensions

Boolean

Booleans are very primitive, they can only be 'true' or 'false'.

Comparisons

Aside from equality `==` and inequality `!=`, booleans also accept the following operators:

& logical AND

```
true  & true  == true
true  & false == false
false & true  == false
false & false == false
```

AND short-circuits expression if left-hand side is false

| logical OR

```
true  | true  == true
true  | false == true
false | true  == true
false | false == false
```

OR short-circuits expression if left-hand side is true

% logical XOR (exclusive OR)

```
true  % true  == false
true  % false == true
false % true  == true
false % false == false
```

XOR always evaluates both sides of the expression

Bang operator

`!!` coerces any type into boolean, like so:

- null -> false
- zero (number) -> false
- non-zero (number) -> true
- empty (text/list/scope/chunk) -> false
- non-empty (text/list/scope/chunk) -> true
- method -> true
- error -> false

logical AND/OR (&, `|`) and conditional flows (`=>`, `?`) will implicitly coerce to boolean

See also

- Boolean prototype extensions
- Flow control

Number

A mathematical concept used to count, measure and do other maths stuff.

Declaration

The `Number` type is implemented as `double`. Here's how to declare a number:

```
a = 5             # number declaration (immutable)
b: Number = 5     # same effect, with type-checking
c: Number = a     # initiating from entry value, also 5
d = 43.14         # with decimals
```

To declare a mutable entry, prepend it with the tilde operator:

```
~ a = 6  # mutable number entry
a += 1   # adds 1 to 'a', yields 7
```

Operating numbers

Numbers accept quite a few operations:

- `==` equal
- `!=` not equal
- `+` plus
- `-` minus
- `*` multiply
- `/` divide
- `%` modulus
- `**` power
- `<` less
- `<=` less or equal
- `>` more
- `>=` more or equal
- `&` logical AND
- `|` logical OR

Caveats

For logical operations and flow control, keep in mind that zero is falsy and non-zero is truthy.

For equality operators, although `0` and `null` are evaluated as falsy, in FatScript they are not the same:

```
0 == null  # false
```

Precision

Although the arithmetic precision of a `IEEE 754 double` is higher, `fry` employs rounding tricks to improve human readability when printing long decimal sequences of nines or zeros as text. Additionally, it uses an epsilon of `1.0e-06` for 'equality' comparisons between numbers.

In 99.999% of use cases, this approach provides both more convenient comparisons and more natural-looking numbers:

```
# Equality epsilon
x = 1.0e-06
x: Number = 0.000001

# Smaller differences are treated as the "same" number by comparison
x == 0.0000015
Boolean: true  # the 0.0000005 difference is ignored
```

Floating-point numbers aren't distributed evenly on the number line. They are dense around 0, and as the magnitude increases, the 'delta' between two expressible values increases:

```
_____________________________________0_____________________________________
+infinity    |     |    |   |  |  | ||| |  |  |   |    |     |    -infinity
```

the biggest contiguous integer is 9,007,199,254,740,992 or 2^53

You can still have much larger numbers, around 10^308, which is:

```
100000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000
```

Bear in mind that if you add 1 to 10^308, no matter how many times you do it, it will always result in the same value! You need to add at least something near 10^293 in a single operation for it to be considered, as the numbers need to be of similar orders of magnitude. To discreetly handle numbers exceeding 2^53, consider using the HugeInt type.

Also, the `infinity` keyword provides a clear, unambiguous representation of values that soar into the realms beyond the largest expressible numbers, approaching the theoretical infinity.

See also

- Number prototype extensions
- Math library

HugeInt

An advanced numerical data type designed to handle very large integers.

Declaration

The `HugeInt` type supports integers up to 4096 bits. Here's how you can declare a `HugeInt`:

```
h = 0x123456789abcdef  # HugeInt declaration
```

`HugeInt` is always expressed in hexadecimal format

Operating HugeInts

`HugeInt` supports a variety of operations, making it versatile for complex calculations:

- `==` equal
- `!=` not equal
- `+` plus
- `-` minus
- `*` multiply
- `/` divide
- `%` modulus
- `**` power
- `<` less
- `<=` less or equal
- `>` more
- `>=` more or equal
- `&` logical AND
- `|` logical OR

Caveats

In FatScript, `HugeInt` is specifically designed as an unsigned type, and thus it can only represent positive values.

Interactions between `HugeInt` and other numeric types, such as Number, are not directly available. To perform such operations, you should convert the value to `HugeInt` using its constructor (available through the prototype extensions).

Precision

`HugeInt` offers high precision for very large integers, essential in fields like cryptography and large-scale computations. This precision remains consistent across its entire range.

```
prime = 0xffffffffffffffc90fdA... # a large prime number
```

Contrary to floating-point numbers, `HugeInt` represents discrete integer values, maintaining consistent precision and spacing throughout its range:

```
0________________________________________
|    |    |    |    |    |    |    |    | overflow
```

the maximum value is 2^4096 - 1, equivalent to a number with 1233 decimal digits or the 0xfff... literal (with 1024 repetitions of the letter f)

`HugeInt` is particularly well-suited for scenarios that demand exact integer arithmetic without rounding errors, especially when dealing with values far beyond the limits of Number type. It is important to ensure that all operations remain within its supported capacity, as exceeding this limit will raise a `ValueError`.

See also

- HugeInt prototype extensions

Text

Texts can hold many characters, and are sometimes referred to as strings.

Declaration

Text entries are declared using quotes:

```
a = 'hello world'        # smart text declaration
a = "hello world"        # raw text declaration
a: Text = 'hello world'  # smart, optionally verbose
```

Manipulating text

Concatenation

In FatScript, you can concatenate, or join, two texts using the `+` operator. This operation connects the two texts into one. For example:

```
x1 = 'ab' + 'cd'  # Outputs 'abcd'
```

Text Subtraction

FatScript also supports a text subtraction operation using the `-` operator. This operation removes a specified substring from the text. For instance:

```
x2 = 'ab cd'
x2 - ' ' == 'abcd'  # Outputs true
```

In the above example, the space character `' '` is removed from the original text `'ab cd'`, resulting in `'abcd'`.

Text Selection

Selection allows you to access specific parts of a text using indices. In FatScript, you can use either positive or negative indices. Positive indices start from the beginning of the text (`0` is the first character), and negative indices start from the end of the text (`-1` is the last character).

> for detailed explanation about the indexing system in FatScript, refer to the section on accessing and selecting items in List

When only one index is passed to the selection function, a single character from the text is selected. When a range is passed to the function, a fragment from the text is selected. This selection is inclusive, meaning that it includes the characters at both the start and end indices, unless using half-open range operator `..<` exclusive on the right-hand side.

Like with lists, accessing items that are out of valid indices will generate an error. For selections, no errors are generated when accessing out-of-bounds indices; instead, an empty text is returned.

```
x3 = 'example'
x3(1)     # 'x'
x3(2, 4)  # 'amp'
x3(..2)   # 'exa'
x3(..<2)  # 'ex'
```

Special characters

Characters such as quotes `'` / `"` can be escaped with backslash `\`.

```
'Rock\'n\'roll'
"Where is \"here\"?"
```

> you only need to escape quotes of same type used as text delimiter

Other supported escape sequences are are:

- backspace `\b`
- new line `\n`

- carriage return `\r`
- tab `\t`
- escape `\e`
- octet in base-8 representation `\ooo`
- backslash itself `\\`

Smart texts

When declared with single quotes `'`, the smart mode is enabled, and interpolation is performed for any code wrapped in curly brackets `{...}`:

```
text = 'world'
interpolated = 'hello {text}'  # outputs 'hello world'
```

> the template is processed in a layer with access to current scope

Note that the use of new lines or other smart texts inside the interpolation template code is not supported, but you can make method calls if you need to compose the result with something more complex.

You can avoid interpolation by escaping the opening bracket:

```
escaped = 'hello \{text}'  # outputs 'hello {text}'
```

Alternatively, you can avoid interpolation by using raw texts.

Raw texts

When declared with double quotes `"` the raw text mode is assumed and interpolation is disabled.

Smart mode vs. raw mode example:

```
'I am smart: {interpolated}'  # using value from previous example
I am smart: hello world       # replacement occurs

"I am raw: {interpolated}"  # brackets are just common characters
I am raw: {interpolated}    # no interpolation occurs
```

Operating texts

- `==` equal
- `!=` not equal
- `+` plus (concatenate)
- `-` minus (removes substring)
- `<` less (alphanumeric)
- `<=` less or equal (alphanumeric)
- `>` more (alphanumeric)
- `>=` more or equal (alphanumeric)
- `&` logical AND (coerced to boolean)
- `|` logical OR (coerced to boolean)

Encoding

FatScript is designed to operate with text encoded in UTF-8. This design choice acknowledges the prevalence of these encoding systems and optimizes the language for broad compatibility.

UTF-8 is a multi-byte encoding system capable of representing any character in the Unicode standard. This universal character encoding scheme uses 8 to 32 bits to represent a character, enabling the depiction of a vast array of symbols from numerous languages and writing systems. Notably, the first 128 characters (0-127) of UTF-8 align precisely with the ASCII set, making any ASCII text a valid UTF-8 encoded string.

In FatScript, the Text data type is a sequence of Unicode characters, inherently encoded in UTF-8, therefore operations such as `text.size`, `text(index)`, and `text(1..4)` will correctly count, access, or slice text irrespective of the complexity of the characters. These operations consider a complete multi-byte UTF-8 character as a single unit, ensuring correct and predictable behavior.

See also

- Text prototype extensions

Method

Methods are recipes that can take arguments to "fill in the blanks".

Definition

A method is anonymously defined with a thin arrow `->`, like so:

```
<parameters> -> <recipe>
```

Parameters can be omitted if none are needed:

```
-> <recipe>  # arity zero
```

To register a method to the scope, assign it to an identifier:

```
<identifier> = <parameters> -> <recipe>
```

Parameters within a method's execution scope are immutable, ensuring that the method's operations do not alter their original state. For mutable behavior, consider passing a scope or utilizing a custom type capable of encapsulating multiple values and states.

Optional parameters

While method signatures typically require a fixed number of mandatory parameters, FatScript supports optional parameters through default values:

```
greet = (message: Text, name: Text = 'World') -> {
  "Hello, {name}, {message}"
}
```

In this example, the `name` parameter is optional, defaulting to 'World' if no argument is provided. This feature allows for more flexible method invocations.

Argument handling

Method calls in FatScript are designed to accept more arguments than required; extra arguments are simply ignored. This behavior is part of the language's design to enhance flexibility and performance.

Auto-return

FatScript uses auto-return, meaning the last standing value is returned:

```
answer: Method = (theGreatQuestion) -> {
  # TODO: explain Life, the Universe and Everything
  42
}

answer("6 x 7 = ?")  # outputs: 42
```

Automatic calls

FatScript introduces a unique feature that simplifies method calls, when no arguments are involved. This feature is known as the "automatic call trick" and it offers several key benefits:

- **Reduced Boilerplate**: Reduces the need for parentheses, making code cleaner and more concise, for zero-parameter methods that act like properties.
- **Dynamic Computation**: Allows for dynamic computation with outputs that can change based on the object's internal or global state.
- **Deferred Execution**: Enables deferred execution, useful in asynchronous programming and complex initialization patterns.

Basic usage

In FatScript, a method defined without parameters is executed "automagically" when referenced:

```
foo = {
  bar = -> 'Hello!'
}

# Both lines below output 'Hello!'
foo.bar()  # explicit call
foo.bar    # automatic call
```

Referencing

To reference a method without triggering the automatic calling feature, you can use the the get syntax:

```
foo('bar')  # yields a reference to foo.bar, without calling it
```

FatScript also offers `self` and `root` keywords to reference methods at the local and global levels, respectively:

```
self('myLocalMethod')
root('myGlobalMethod')
```

Opting out of automatic calls

The tilde ~ operator allows you to bypass the automatic call feature, providing flexibility in method handling:

```
# Both lines below fetch the method reference, without calling it
foo.~bar
~ myMethod
```

Or you can simply wrap the method call into yet another (anonymous) method:

```
-> foo.bar
```

Passing methods as arguments

There's an important exception when it comes to passing methods as arguments, specifically in the case of a local method:

```
another(bar)  # passes `bar` as a reference, without executing it
```

> however, this does not apply with chaining: `another(foo.bar)` passes the result of `bar`, not the reference

In this case, to pass the value resulting of the local method `bar`, an explicit call must be made:

```
another(bar())
```

> this behavior might seem counterintuitive, but it is extremely useful in various use cases, such as when passing methods to [reduce](), to an [asynchronous task](), to a [mapping]() operation etc.

Implicit argument

A convenience offered by FatScript is the ability to reference a value passed to the method without explicitly specifying a name for it. In this case, the implicit argument is represented by the underscore _.

Here's an example that illustrates the use of implicit argument:

```
double = -> _ * 2
double(3)  # output: 6
```

You can use an implicit argument whenever you need to perform a simple operation on a single parameter without assigning a specific name to it, but note that the method must have arity zero to trigger it.

See also

- [Method prototype extensions]()

List

Lists are ordered collections of items of the same type, accessed by index.

Definition

Lists are defined with square brackets `[]`, like so:

```
list: List/Text = [ 'apple', 'pizza', 'pear' ]
```

Lists do not allow mixing of types. The type of a list is determined by the first item added to it, consequently, empty lists are untyped.

Lists skip empty positions, so an item that evaluates to `null` is ignored:

```
a = 1
c = 3
[ a, b, c ]  # outputs: [ 1, 3 ] (b is skipped over)
```

Access

Individual items

List items can be accessed individually with zero-based index call:

```
list(0)  # 'apple'
list(2)  # 'pear'
```

Negative values will index backwards, starting from -1 as the last item:

```
list(-1)  # 'pear'
```

Accessing items that are out of valid indices will generate an error:

```
         0        1        2       > 2
Error [ 'apple', 'pizza', 'pear' ] Error
 < -3       -3       -2      -1
```

Selections

Indexes for start and end work exactly the same as when accessing individual items, so negatives count from the last item and can be regressive. However, when using ranges, no errors are generated when accessing out-of-bounds indices; instead, an empty list is returned.

```
list(0..0)   # [ 'apple' ]
list(4..8)   # []
list(1..-1)  # [ 'pizza', 'pear' ]
```

One index can be left blank, and the start from the first or the end at the last item is assumed:

```
list(..1)   # [ 'apple', 'pizza' ]
list(1..)   # [ 'pizza', 'pear' ]
```

Deprecation warning

The option to passing a second argument (using a comma, and not dot-dot) to perform a selection is being deprecated starting from version 3.x.x:

```
list(0, 0)   # [ 'apple' ]          # deprecated syntax
list(4, 8)   # []                   # deprecated syntax
list(1, -1)  # [ 'pizza', 'pear' ]  # deprecated syntax
```

Nested lists

A matrix can be used and accessed like so:

```
matrix = [
  [ 1, 2, 3 ]
  [ 4, 5, 6 ]
]

matrix(1)(0)  # yields 4 (1: second line, then 0: first index)
```

for simplicity, the example uses a 2D matrix, but could be n-dimensional

Operations

- `==` equal
- `!=` not equal
- `+` addition (concatenation effect)
- `-` subtraction (difference effect)
- `&` logical AND
- `|` logical OR

logical AND/OR evaluate empty lists as `false`, otherwise `true`

List addition (concatenation)

The list addition operation allows you to combine two lists into a new list:

```
x = [1, 2, 2, 3]
y = [3, 3, 4, 4]

x + y  # result: [1, 2, 2, 3, 3, 3, 4, 4]
```

In this case, using the addition operator `+` to merge lists `x` and `y`, the elements from both lists are combined into a single list. The order of the elements in the resulting list is determined by the order in which the lists were added.

there is no removal of duplicate elements during the concatenation

Quick-append

For better performance, you can take advantage of `+=` operator, e.g.:

```
~ list += [ value ]  # faster

# same effect as
~ list = []
list = list + [ value ]  # concatenation (slower)
```

Another detail of the `+=` operator, which also applies to other types, is the automatic initialization by omission, where if the entry has not yet been declared previously, it acts as a simple assignment.

List subtraction (difference)

The list subtraction operation allows you to remove elements from the second operand that are present in the first operand, resulting in a list containing only unique values:

```
x = [ 1, 2, 2, 3 ]
y = [ 3, 3, 4, 4 ]

x - y  # result: [ 1, 2 ]
y - x  # result: [ 4 ]
```

In this case, when we subtract the list `y` from the list `x`, the elements with the value 3 are removed because they are present in both lists. The result is the list `[1, 2]`. Similarly, when we subtract the list `x` from the list `y`, the only remaining element is the value 4.

only exactly identical values are removed during the subtraction

See also

- [List prototype extensions]()
- [Mapping over a List]()

Scope

A scope is akin to a dictionary, where keys are associated with values.

Definition

Scopes are defined using curly brackets `{}`, as shown below:

```
myCoolScope = {
  place = 'here'
  when = 'now'
}
```

Scopes store entries in alphabetical order, a characteristic that becomes apparent when mapping over a scope.

Access

There are three ways you can directly access entries inside a scope.

Dot syntax

```
myCoolScope.place  # output: 'here'
```

Get syntax

```
# assuming prop = 'place'
myCoolScope(prop)  # output: 'here'
```

In both methods, if the property is not present, `null` is returned. If the outer scope is not found, an error is raised.

Optional chaining syntax

Use the question-dot `?.` operator to safely chain potentially non-existent outer scopes:

```
nonExisting?.prop  # returns null
```

The optional chaining syntax does not raise an error when the outer scope is `null`.

Operations

- `==` equal
- `!=` not equal
- `+` addition (merge effect)
- `-` subtraction (difference effect)
- `&` logical AND
- `|` logical OR

logical AND/OR evaluate empty scopes as `false`, otherwise `true`

Scope addition (merge)

The second operand acts as a patch for the first operand:

```
x = { a = 1, b = 3 }
y = { b = 2 }

x + y  # results in { a = 1, b = 2 }
y + x  # results in { a = 1, b = 3 }
```

values from the second operand replace those from the first

Scope subtraction (difference)

Subtraction removes elements from the first operand that are identical to those in the second operand:

```
x = { a = 1, b = 3 }
y = { a = 1 }

x - y  # results in { b = 3 }
```

> only values that are exactly identical are removed

Scoped Blocks

Scoped Blocks in FatScript allow for executing statements within the context of a specific scope:

```
object.{
  # Statements executed in the context of 'object'
}
```

Here, `object` is the target scope. Within the block, you can directly access and modify `object`'s properties.

Features

- **Isolation**: entries declared within a Scoped Block are local to that block and do not affect the outer scope
- **Outer Scope Access**: Scoped Blocks can access entries from the outer scope

Example

```
x = {}

x.{
  a = 5      # 'a' is now a property of 'x'
  b = a + 3  # 'b' is also a property of 'x'
}
```

Scope interactions

FatScript uses sophisticated mechanisms for managing variables across different scopes, leveraging concepts of lexical scoping and shadowing to provide powerful programming capabilities. This section explores these mechanisms, including assignment nuances, increment/decrement behaviors, and the innovative use of the += operator for boolean toggling.

Assignment

The assignment operator (=) copies values from outer scopes into current scope, defining a new value:

```
~ n = 1
x = {}
x.{ ~ n = n }  # now x.n == 1, and x.n is independent from root.n
x.{ c = n }    # has similar effect, however 'c' is immutable
```

> the same concept applies to code running on a method scope

Caveat

Using `~ n = n + 1` inside a block or method adds a new 'n' in the current scope, initialized with the value of `n + 1` from the nearest enclosing scope, without altering the outer `n`.

Incrementing and decrementing

Increment (+=) and decrement (-=) operations, interact with variable scoping in a different way. These operations search for the nearest instance of a variable, starting from the current scope and moving outward recursively, and then modify that instance directly.

```
~ outerN = 1
fn = -> {
  outerN += 1  # targets and increments 'outerN' in the outer scope
}
```

Auto-initialization with +=

FatScript also provides a special behavior regarding increment operator (+=). If the entry doesn't exist, increment works as a regular assignment as if you had written the following for `n += 1`:

```
n == Void ? n = 1 : n += 1
```

The auto-initialization feature can be particularly useful when used in combination with [dynamic entries] for dynamic programming.

> this feature is exclusively available for increment operator, decrement can't initialize non-existent values

Boolean toggling with +=

Generally, booleans don't allow addition operations. FatScript, however, extends the `+=` operator's functionality to boolean types, allowing for an intuitive toggle mechanism within inner scopes.

The expression `flag += !flag` effectively toggles the boolean value, even when `flag` is defined in an outer scope.

> in the particular case of booleans, the only distinction between `=` and `+=` is scoping

Other compound assignment operators

Similarly, other compound assignment operations such as `*=`, `/=`, `%=`, and `**=` are supported by numeric types and respect the same scoping rules that apply to increment and decrement operations.

See also

- Dynamic entries
- Scope prototype extensions
- Mapping over a scope

Error

There is great wisdom in expecting the unexpected too.

Default subtypes

While some generic errors like syntax issues, invalid imports etc. are raised with the base Error type, some others are subtyped.

See the definitions in the error prototype extensions.

Declaration

Errors can also be declared explicitly; you must use the type constructor:

```
_ <- fat.type.Error

Error('an error has ocurred')  # raises a generic error

MyMistake = Error
MyMistake('another error has ocurred')  # raises a MyMistake subtype error
```

Comparisons

Errors always evaluate as falsy:

```
Error() ? 'is truthy' : 'is falsy'  # is false
```

Errors are comparable to their type:

```
Error() == Error  # true
```

read also about type comparison syntax

A naive way of handling errors could be:

```
_ <- fat.console
# handling the returned error
maybeFail() <= Error => log('an error has happened')
_                    => log('success')
```

this only works if option `-e / continue on error` is set

Another naive way to deal with errors, but one that always works, is to use a default operation:

```
maybeFail() ?? log('an error occurred')
```

Although the naive approach may work, the proper way to deal with errors is by setting an error handler using the `trapWith` method found in the failure library.

See also

- Failure library
- Error prototype extensions

Chunk

Chunks are just binary blocks of data.

Declaration

Chunks cannot be declared explicitly; you must use the type constructor and apply one of the following strategies:

```
_ <- fat.type.Chunk

Chunk(null)            # Void -> (empty chunk)
Chunk(true)            # Boolean -> '\001'
Chunk(65)              # Number -> 'A'
Chunk('ABC')           # Text -> 'ABC'
Chunk([ 65, 66, 67 ])  # List/Number -> 'ABC'
```

> numbers are expected to be valid byte values (0-255), otherwise an error is raised

Manipulating Chunks

Concatenation

In FatScript, you can concatenate, or join, two chunks using the `+` operator. For example:

```
abCombined = chunkA + chunkB
```

Chunk Selection

Selection allows access to specific parts of a chunk using indices. FatScript supports both positive and negative indices. Positive indices start from the beginning of the chunk (with `0` as the first byte), while negative indices start from the end (`-1` is the last byte).

> for detailed explanation about the indexing system in FatScript, refer to the section on accessing and selecting items in List

Selecting with one index retrieves a single byte from the chunk (as number). Using a range of bytes, selects a fragment inclusive of both start and end indices, except when using the half-open range operator `..<`, which is exclusive on the right-hand side.

Accessing indices outside the valid range will generate an error for individual selections. For range selections, out-of-bounds indices result in an empty chunk.

```
x3 = Chunk('example')
x3(1)     # 120 (ASCII value of 'x')
x3(..2)   # new Chunk containing 3 bytes (corresponding to 'exa')
```

Comparisons

Chunk equality `==` and inequality `!=` comparisons are supported.

See also

- Chunk prototype extensions

Flow control

Move along in a continuous stream of decisions that should be made.

Fallback

Default or nullish coalescing operations, are defined with double question marks `??` and work the following way:

```
<maybeNullOrFailedExpression> ?? <fallbackValue>
```

In case the left-hand side is not `null` nor `Error`, then it's used; otherwise, the fallback value is returned.

> similarly you can use the nullish coalescing assign operator `??=`

If

`If` statements are defined with a question mark `?`, like so:

```
<condition> ? <response>
```

> as there is no alternative `null` is returned if condition is not met

If-Else

`If-Else` statements are defined with a question mark `?` followed by a colon `:`, like so:

```
<condition> ? <response> : <alternativeResponse>
```

To use multiline `If-Else` statements, wrap the response in curly brackets `{...}` like so:

```
<condition> ? {
  <response>
} : {
  <alternativeResponse>
}
```

Cases

Cases are defined with the thick arrow `=>` and are automatically chained, creating an intuitive and streamlined syntax similar to a switch statement without the possibility of fall-through. This allows for unrelated conditions to be mixed together, ultimately resulting in a more concise if-else-if-else structure:

```
<condition1> => <responseFor1>
<condition2> => <responseFor2>
<condition3> => <responseFor3>
...
```

Example:

```
choose = (x) -> {
  x == 1 => 'a'
  x == 2 => 'b'
  x == 3 => 'c'
}

choose(2)  # 'b'
choose(8)  # null
```

To provide a default value for your method, you can add a catch-all case using an underscore _ at the end of the sequence:

```
choose = (x) -> {
  x == 1 => 'a'
  x == 2 => 'b'
  x == 3 => 'c'
  _      => 'd'
}
```

```
choose(2)  # 'b'
choose(8)  # 'd'
```

For more complex scenarios, you can use blocks as outcomes for each case:

```
...
  condition => {
    # do something
    'foo'
  }
  _ => {
    # do something else
    'bar'
  }
...
```

Cases must end in a catch-all case _ or end of block. The most effective use of Cases is within methods at the bottom of the method body.

While it's possible to add nested Cases, it's best to avoid overly complex constructions. This makes code harder to follow and likely misses the point of using this feature.

It may be more appropriate to extract that logic into a separate method. FatScript encourages developers to split logic into distinct methods, helping to prevent spaghetti code.

Switch

The Switch operator is denoted by the >> symbol, which guides the flow of control based on the value's match against a series of cases:

Syntax:

```
<value> >> {
  <caseValue1> => <responseFor1>
  <caseValue2> => <responseFor2>
  ...
  _ => <defaultResponse>
}
```

Each case in the Switch block is evaluated in order until a match is found and the result of the matching case is returned:

```
choose = -> _ >> {
  1 => 'one'
  2 => 'two'
  3 => 'three'
  _ => 'other'
}

choose(2)  # 'two'
choose(4)  # 'other'
```

Switch cases can also involve expressions, allowing for dynamic matching:

```
evaluate = (x, y) -> x >> {
  y + 1 => 'just above y'
  y - 1 => 'just below y'
  _     => 'not directly around y'
}

evaluate(5, 4)  # 'just above y'
evaluate(3, 4)  # 'just below y'
evaluate(7, 4)  # 'not directly around y'
```

Loops

Repeat, repeat, repeat, repeat, repeat...

Base syntax

All loops are build with an "at" sign `@`, like so:

```
<expression> @ <loopBody>
```

While loop

The loop body will execute while the expression evaluates to:

- true
- non-zero number
- non-empty text

The execution will terminate when the expression evaluates to:

- false
- null
- zero number
- empty text
- error

For example, this loop prints numbers 0 to 3:

```
_ <- fat.console

~ i = 0

(i < 4) @ {
  log(i)
  i += 1
}
```

Mapping syntax

You can map over ranges, lists and scopes with a mapper, like so:

```
<range|collection> @ <mapper>
```

A new list is generated based from the return values of the mapper.

Mapping over a range

Using range operator `..` the mapper will receive a number as input sequentially from the left bound to the right bound:

```
4..0 @ num -> num + 1  # returns [ 5, 4, 3, 2, 1 ]
```

> range syntax is inclusive on booth sides, e.g. 0..2 yields 0, 1, 2

There is also half-open range operator `..<` exclusive on the right-hand side.

> caveat: half-open range won't work with reverse direction, always needs to be from the minimum to maximum

Mapping over a list

The mapper will receive items in order (from left to right):

```
[ 3, 1, 2 ] @ item -> item + 1  # returns [ 4, 2, 3 ]
```

Mapping over a scope

The mapper will receive the names (keys) of the entries stored in the scope in alphabetical order:

```
{ c = 3, a = 1, b = 2 } @ key -> key  # yields [ 'a', 'b', 'c' ]
```

> on the examples we have used list and scope literals, but an entry or call that evaluates to a list or a scope will have the same effect

To access entries in a scope, you refer to it by name, but in this case, it needs to be defined in the outer scope, for example:

```
myScope = { c = 3, a = 1, b = 2 }
myScope @ key -> myScope(key)  # returns [ 1, 2, 3 ]
```

FatScript uses an intelligent caching feature that prevents this syntax from generating additional effort to search for the current element in the scope while mapping.

Libraries

Let's talk about the sweet fillings baked into FatScript: the libraries!

Standard libraries

Essentials

These are the fundamental libraries you would expect to be available in a programming language, providing essential functionality:

- async - Asynchronous workers and tasks
- color - ANSI color codes for console
- console - Console input and output operations
- curses - Terminal-based user interface
- enigma - Cryptography, hash and UUID methods
- failure - Error handling and exception management
- file - File input and output operations
- http - HTTP handling framework
- math - Mathematical operations and functions
- recode - Data conversion between various formats
- sdk - Fry's software development kit utilities
- system - System-level operations and information
- time - Time and date manipulation

Type Package

This package extends the features of FatScript's native types:

- Void
- Boolean
- Number
- HugeInt
- Text
- Method
- List
- Scope
- Error
- Chunk

Extra package

Additional types implemented in vanilla FatScript:

- Date - Calendar and date handling
- Duration - Millisecond duration builder
- HashMap - Quick key-value store
- Logger - Logging support
- Memo - Generic memoization utility
- Option - Encapsulation of optional value
- Param - Parameter presence and type verification
- Sound - Sound playback interface
- Storable - Data store facilities

Import-all shorthand

If you want to make all of them available at once, you can simply do the following, and all that good stuff will be available to your code:

```
_ <- fat._
```

While this feature can be convenient when experimenting on the REPL, be aware that it brings in all the library's constants and method names, potentially polluting your global namespace.

fat.std

Alternatively, import the "standard" library, which imports all types (including those from the extra package), as well as named imports from all other packages, like this:

```
_ <- fat.std
```

This is equivalent to:

```
_       <- fat.type._
_       <- fat.extra._
async   <- fat.async
color   <- fat.color
console <- fat.console
curses  <- fat.curses
enigma  <- fat.enigma
failure <- fat.failure
http    <- fat.http
file    <- fat.file
math    <- fat.math
recode  <- fat.recode
sdk     <- fat.sdk
system  <- fat.system
time    <- fat.time
```

Note that importing everything in advance can add unnecessary overhead to the startup time of your program, even if you only need to use a few methods.

As a best practice, consider importing only the specific modules you need, with named imports. This way, you can keep your code clean and concise, while minimizing the risk of naming conflicts or performance issues.

Hacking and more

Under the hood, libraries are built using embedded commands. To gain a deeper understanding and explore the inner workings of the interpreter, dive into this more advanced topic.

async

Asynchronous workers and tasks

Import

```
_ <- fat.async
```

Types

The `async` library introduces the `Worker` type.

Worker

The `Worker` is a simple wrapper around an asynchronous operation.

Constructor

Name	Signature	Brief
Worker	(task: Method, wait: Number)	Builds a Worker in standby mode

The `Worker` constructor takes two arguments:

- **task**: The method to be executed asynchronously (the method may not take arguments directly, but you may curry those in using two arrows on the definition `-> ->`).
- **wait** (optional): A timeout in milliseconds. If the task does not finish within this time, it is cancelled.

Prototype members

Name	Signature	Brief
start	(): Worker	Begins the task
cancel	(): Void	Cancels the task
await	(): Worker	Waits for task completion
isDone	(): Boolean	Checks if the task has completed
hasStarted	Boolean	Set by start method
hasAwaited	Boolean	Set by await method
isCanceled	Boolean	Set by cancel method
result	Any	Set by await method

Standalone Methods

Name	Signature	Brief
atomic	(op: Method): Any	Executes the operation atomically
selfCancel	(): *	Terminates the execution of the thread
processors	(): Number	Get the number of processors

Usage Notes

`Worker` instances are mapped to system threads on a one-to-one basis and get executed as per the system's scheduling. This implies that their execution may not always be immediate. To wait for the result of a `Worker`, employ the `await` method.

Unlike in other contexts, in asynchronous code, the `task: Method` executes without access to the scope in which it is created. It can only access properties that have been 'curried' `-> ->` into its execution scope or those that are directly accessible in the global scope.

> to keep maximum performance, avoid using [text interpolation]() within asynchronous tasks

Examples

```
async <- fat.async
math  <- fat.math
time  <- fat.time

# Define a slow task
slowTask = (seconds: Number): Text -> -> {
  time.wait(seconds * 1000)
  'done'
}

# Start the task as a Worker
worker = Worker(slowTask(5)).start

# Get the worker result
result1 = worker.await.result  # blocks until task is done

# Start a task with timeout
task = Worker(slowTask(5), 3000).start  # task should timed out

# Get the task result
result2 = task.await.result  # blocks until task is done or timeout occurs
```

> the `await` method will raise `AsyncError` if the task times out before completion

atomic

The `atomic` wrapper is a critical tool for ensuring thread safety and data integrity in concurrent programming. When multiple workers or asynchronous tasks access and modify shared resources, race conditions can occur, leading to unpredictable and erroneous outcomes. The `atomic` operation addresses this issue by guaranteeing that the method it wraps is executed atomically. This means the entire operation is completed as a single, indivisible unit, with no possibility of other threads intervening partway through for the same operation. This is particularly important for operations such as incrementing a counter, updating shared data structures or files, or performing any action where the order of execution matters:

```
async.atomic(-> file.append(logFile, line))
```

While `atomic` operations are a powerful tool for ensuring consistency, it's important to be mindful of the potential for contention it introduces. Contention occurs when multiple threads or tasks attempt to execute an operation simultaneously, leading to potential performance bottlenecks as each thread waits its turn. Overuse or unnecessary use of `atomic` operations can significantly degrade the performance of your application by reducing concurrency. Keep only the critical section of code that absolutely requires atomicity enclosed as an `atomic` operation.

> under the hood, atomic operations are fundamentally guarded by a single global `mutex`

Async in Web Build

When using `fry` built with Emscripten (for example, when using FatScript Playground), the platform's limited support for multi-threading affects the `Worker` implementation. To maximize cross-platform code compatibility, `Worker` tasks execute inline and block the main thread when the `start` method is called. This approach compromises the advantages of asynchronous execution but allows a consistent implementation across platforms in many scenarios.

See also

- [Time library](#)

color

ANSI color codes for console

Import

```
_ <- fat.color
```

Constants

- black, 0
- red, 1
- green, 2
- yellow, 3
- blue, 4
- magenta, 5
- cyan, 6
- white, 7
- bright.black, 8
- bright.red, 9
- bright.green, 10
- bright.yellow, 11
- bright.blue, 12
- bright.magenta, 13
- bright.cyan, 14
- bright.white, 15

Methods

Name	Signature	Brief
detectDepth	(): Number	Get console color support
to8	(xr: Any, g: Number = ø, b: Number = ø)	Convert RGB to 8-color mode
to16	(xr: Any, g: Number = ø, b: Number = ø)	Convert RGB to 16-color mode
to256	(xr: Any, g: Number = ø, b: Number = ø)	Convert RGB to 256-color mode

Usage Notes

to8, to16 and to256

The parameter `xr` can be an optional text representing the color in HTML format. For example, it can be provided as 'fae830' or '#fae830' (yellow):

```
color   <- fat.console
console <- fat.console

console.log('hey', color.to16('fae830'))
console.log('hey', color.to256('fae830'))
```

However, if `xr` is a number between 0 and 255 representing `r`, then the `g` and `b` parameters will be required:

```
console.log('hey', color.to256(250, 232, 48))  // same result
```

these methods may produce approximations of the original color in 8, 16 or 256 depths and not the exact true color

See also

- [Console library]
- [Curses library]
- [256 Colors]

console

Console input and output operations

Import

```
_ <- fat.console
```

Methods

Name	Signature	Brief
log	(msg: Any, fg: Number = ø, bg: Number = ø): Void	Print msg to stdout, with newline
print	(msg: Any, fg: Number = ø, bg: Number = ø): Void	Print msg to stdout, without newline
stderr	(msg: Any, fg: Number = ø, bg: Number = ø): Void	Print msg to stderr, with newline
input	(msg: Any, mode: Text = ø): Text	Print msg and return input of stdin
flush	(): Void	Flush stdout buffer
cls	(): Void	Clear stdout using ANSI escape codes
moveTo	(x: Number, y: Number): Void	Move cursor using ANSI escape codes
isTTY	(): Boolean	Check if stdout a terminal device
isTty	(): Boolean	DEPRECATED (will be removed in `3.x.x`)
showProgress	(label: Text, fraction: Number): Void	Render progress bar, fraction 0 to 1

> the methods `log`, `stderr` and `input` ensure thread safety in asynchronous scenarios

Usage Notes

output

By default, `stdout` and `stderr` both print to the console. The foreground color (fg) and background color (bg) parameters are optional.

> colors are automatically suppressed if the output buffer is not a TTY

input

The optional `mode` parameter accepts the following values:

- 'plain', plain input (no readline cursor, no history)
- 'quiet', like plain mode, but without feedback
- 'secret', special mode for password input
- `null` (default), with readline and input history

See also

- Color library
- Curses library

curses

Terminal-based user interface

> although the inspiration is acknowledged, FatScript has it's own way of approaching terminal UI which differs in many ways from the original curses library

Import

```
_ <- fat.curses
```

Methods

Name	Signature	Brief
box	(p1: Scope, p2: Scope): Void	Draw square from pos1 to pos2
fill	(p1: Scope, p2: Scope, p: Text = ' '): Void	Fill from pos1 to pos2 with p
clear	(): Void	Clear screen buffer
refresh	(): Void	Render screen buffer
getMax	(): Scope	Return screen size as x, y
printAt	(pos: Scope, msg: Any, width: Number = ø): Void	Print msg at { x, y } pos
makePair	(fg: Number = ø, bg: Number = ø): Number	Create a color pair
usePair	(pair: Number): Void	Apply color pair
frameTo	(cols: Number, rows: Number)	Align view to screen center
readKey	(): Text	Return key pressed
readText	(pos: Scope, width: Number, prev: Text = ø): Text	Start a text box input
flushKeys	(): Void	Flush input buffer
endCurses	(): Void	Exit curses mode

> positions (pos) are of form { x: Number, y: Number }

> the methods in this library **do not ensure** thread safety in asynchronous scenarios, use either the main thread **or** a single [worker](#) to render console updates

Usage Notes

Any method of this library, except `getMax` and `endCurses`, will start curses mode if not yet started. Note that methods such as `log`, `stderr` and `input` from [console](#) library will implicitly call `endCurses`. However, `moveTo`, `print` and `flush` will not change the output mode, and can be paired with curses methods, which can be useful in some circumstances.

The letters `x` and `y` stand for column and row respectively when calling `printAt`, where (0, 0) is the upper-left corner and the result of `getMax` is the just the first coordinate outside the lower-right corner.

> special characters on curses only work if a UTF-8 [locale](#) can be set

makePair

You can import the [color](#) library to use color names and create a combination of foreground and background (pair). Pass `null` to apply the default color to the desired parameter.

usePair

The input of this method should be a color pair created with `makePair` method. It leaves this pair enabled until you call it again with a different pair.

readKey

This method is non-blocking and returns `null` if `stdin` is empty, otherwise it will return one character at a time.

Special keys may be detected and return keywords such as:

- arrow keys:
  - up
  - down
  - left
  - right
- edit keys:
  - delete
  - backspace
  - enter
  - space
  - tab
  - backTab (shift+tab)
- control keys:
  - pageUp
  - pageDown
  - home
  - end
  - insert
  - esc
- other:
  - resize (terminal window was resized)

the correct detection of keys can depend on the context or platform

readText

Enters text capture mode using an area demarcated by position and width of the text box. If the text is larger than the space, an automatic text scroll is performed. The full text is returned when `enter` or `tab` is pressed, however, if `esc` is pressed, `null` is returned.

See also

- Color library
- Console library

enigma

Cryptography, hash and UUID methods

Import

```
_ <- fat.enigma
```

Methods

Name	Signature	Brief
getHash	(msg: Text): Number	Get 32-bit hash of text
genUUID	(): Text	Generate a UUID (version 4)
genKey	(len: Number): Text	Generate random key
derive	(secret: Text): Text	Key derivation function
encrypt	(msg: Text, key: Text = ø): Text	Encrypt msg using key
decrypt	(msg: Text, key: Text = ø): Text	Decrypt msg using key

`derive` is deterministic and uses the Base64 alphabet for a 32 chars output

Usage Notes

You can omit or pass a blank key `''` for using the default key.

Heads Up!

Although `enigma` makes encrypted text "non-human-readable", this schema is not cryptographically safe! DO NOT use it alone to protect data!

If paired with a custom key that is not stored alongside the message it may offer some data protection.

UUID method conformance

A UUID, or Universally Unique Identifier, is a 128-bit number used to identify objects or entities in computer systems. The provided implementation generates random UUIDs as text that follow the format of version 4 RFC 4122 specification, but does not strictly adhere to the required cryptographically secure randomness. In practice, the collision risk has an extremely low probability and is very unlikely to occur, and for most applications can be considered good enough.

failure

Error handling and exception management

Import

```
_ <- fat.failure
```

Methods

Name	Signature	Brief
trap	(): Void	Apply generic error handler
trapWith	(handler: Method): Void	Set a handler for errors in context
untrap	(): Void	Unset error handler in context

Usage Notes

When an error is created if an error handler is found, seeking from inner to outer execution context, the handler wrapping the failure is automatically invoked with that error as argument, and the calling context is exited with return value of the error handler.

> it's not possible to set a handler for the global scope

trapWith

This method binds an error handler to the context of the calling site, e.g. when used inside a method it will only protect the logic executed inside the body of that method.

Example

Define an error handler that prints the error and exits:

```
console <- fat.console
system  <- fat.system
sdk     <- fat.sdk

simpleErrorHandler = (error) -> {
  console.log(error)
  sdk.printStack(10)
  system.exit(system.failureCode)
}
```

Finally, use `trapWith` method to assign the error handler:

```
failure <- fat.failure
failure.trapWith(simpleErrorHandler)
```

Trap it!

You can handle expected errors or pass through the unexpected:

```
failure <- fat.failure
_       <- fat.type.Error

MyError = Error

errorHandler = -> _ >> {
  MyError => 0  # handle (expected)
  _       => _  # pass through (unexpected)
}

unsafeMethod = (n) -> {
  failure.trapWith(errorHandler)
```

```
  n < 10 ? MyError('arg is less than ten')
  n - 10
}
```

In this case the program will not crash if you call `unsafeMethod(5)`, but if you comment out the `trapWith` line, you will see it crashing with `MyError`.

See also

- Error (syntax)
- Error prototype extensions
- Flow control

file

File input and output operations

Import

```
_ <- fat.file
```

Type contributions

Name	Signature	Brief
FileInfo	(modTime: Epoch, size: Text)	File metadata

Methods

Name	Signature	Brief
basePath	(): Text	Extract path where app was called
exists	(path: Text): Boolean	Check file exists on provided path
read	(path: Text): Text	Read file from path (text mode)
readBin	(path: Text): Chunk	Read file from path (binary mode)
write	(path: Text, src): Boolean	Write src to file and return success
append	(path: Text, src): Boolean	Append to file and return success
remove	(path: Text): Boolean	Remove file and return success
isDir	(path: Text): Boolean	Check if path is a directory
mkDir	(path: Text, safe: Boolean)	Create a directory
lsDir	(path: Text): List	Get list of files in a directory
stat	(path: Text): FileInfo	Get file metadata

Usage Notes

read

On exception:

- logs error to `stderr`
- returns `null`

> `read` cannot see builtin "files", but `readLib` from SDK lib can

write/append

Exceptions:

- logs error to `stderr`
- returns `false`

mkDir

If `safe` is set to `true`, the directory gets 0700 permission instead of default 0755, which is less protected.

See also

- Recode library

http

HTTP handling framework

Import

```
_ <- fat.http
```

Route

A route is a structure used to map HTTP methods to certain path patterns, specifying what code should be executed when a request comes in. Each route can define a different behavior for each HTTP method (`POST`, `GET`, `PUT`, `DELETE`).

Constructor

Name	Signature	Brief
Route	(path: Text, post: Method, get: Method, put: Method, delete: Method)	Constructs a Route object

> each implemented method receives an HttpRequest as argument and shall return an HttpResponse object

HttpRequest

An `HttpRequest` represents an HTTP request message. This is what your server receives from a client when it makes a request to your server.

Constructor

Name	Signature	Brief
HttpRequest	(method: Text, path: Text, params: Scope, headers: List/Text, body: Any)	Constructs an HttpRequest object

HttpResponse

An `HttpResponse` represents an HTTP response message. This is what a server sends back to the client in response to an HTTP request.

Constructor

Name	Signature	Brief
HttpResponse	(status: Number, headers: List/Text, body: Any)	Constructs an HttpResponse object

Methods

Name	Signature	Brief
setHeaders	(headers: List): Void	Set headers of requests
post	(url: Text, body, wait): HttpResponse	Create/post body to url
get	(url: Text, wait): HttpResponse	Read/get from url
put	(url: Text, body, wait): HttpResponse	Update/put body to url
delete	(url: Text, wait): HttpResponse	Delete on url
setName	(name: Text): Void	Set user agent/server name
verifySSL	(enabled: Boolean): Void	SSL configuration (client mode)
setSSL	(certPath: Text, keyPath: Text): Void	SSL configuration (server mode)
listen	(port: Number, routes: List/Route)	Endpoint provider (server mode)

> `body: Any` and `wait: Number` are always optional parameters, being that if `body` does not fall under `Text` or `Chunk`, it will be automatically converted to JSON during the send process, and `wait` is the maximum waiting time and the default is 30,000ms (30 seconds)

> `verifySSL` is enabled by default for the client mode

setSSL may not be available, case the system doesn't have OpenSSL

Usage Notes

Client mode

In the `HttpResponse.body`, you may need to explicitly parse a JSON response to `Scope` using the `fromJSON` method. To post a native type as JSON, you can encode it using the `toJSON` method; however, this is not strictly necessary, as it will be done implicitly. Both methods are available in the [fat.recode] library.

If headers are not set, the default `Content-Type` header for `Chunk` will be `application/octet-stream`, for `Text` will be `text/plain; charset=UTF-8` and for other types, it will be `application/json; charset=UTF-8` (due to implicit conversion).

You can set custom request headers like so:

```
http <- fat.http

url = ...
token = ...
body = ...

http.setHeaders([
  "Accept: application/json; charset=UTF-8"
  "Content-Type: application/json; charset=UTF-8"
  "Authorization: Bearer " + token  # custom header
])

http.post(url, body)
```

setting headers will completely replace previous list with new list

When performing async requests, you may need to call `setHeaders`, `setName`, and configure `verifySSL` within each Worker, as these settings are local to each thread.

Server mode

Handling HTTP Responses

The FatScript server automatically handles common HTTP status codes such as 200, 400, 404, 405, 500, and 501. Being 200 the default when constructing an `HttpResponse` object.

In addition to the common status codes, you can also explicitly return other status codes, such as 201, 202, 203, 204, 205, 301, 401, and 403, by specifying the status code in the `HttpResponse` object, for example: `HttpResponse(status = 401)`. In all cases, where applicable, the server provides default plain text bodies. However, you have the option to override these defaults and provide your own custom response bodies when necessary.

By automatically handling these status codes and providing default response bodies, the FatScript server simplifies the development process while still allowing you to have control over the response content when needed.

if the status code doesn't belong to any of the above, the server will return a 500 code

See an example of a simple file HTTP server:

```
_    <- fat.type.Text
file <- fat.file
http <- fat.http
{ Route, HttpRequest, HttpResponse } = http

# adapt to content location
basePath = '/home/user/contentFolder'

# restrict to some extensions only
getContentType = (path: Text): Text -> {
  ext2 = path(-3..).toLower
  ext3 = path(-4..).toLower
  ext4 = path(-5..).toLower
```

```
  ext4 == '.html' => 'Content-Type: text/html'
  ext3 == '.htm'  => 'Content-Type: text/html'
  ext2 == '.js'   => 'Content-Type: application/javascript'
  ext4 == '.json' => 'Content-Type: application/json'
  ext3 == '.css'  => 'Content-Type: text/css'
  ext2 == '.md'   => 'Content-Type: text/markdown'
  ext3 == '.xml'  => 'Content-Type: application/xml'
  ext3 == '.csv'  => 'Content-Type: text/csv'
  ext3 == '.txt'  => 'Content-Type: text/plain'
  ext4 == '.svg'  => 'Content-Type: image/svg+xml'
  ext3 == '.rss'  => 'Content-Type: application/rss+xml'
  ext4 == '.atom' => 'Content-Type: application/atom+xml'
  ext3 == '.png'  => 'Content-Type: image/png'
  ext3 == '.jpg'  => 'Content-Type: image/jpeg'
  ext4 == '.jpeg' => 'Content-Type: image/jpeg'
  ext3 == '.gif'  => 'Content-Type: image/gif'
  ext3 == '.ico'  => 'Content-Type: image/icon'
}

routes: List/Route = [
  Route(
    '*'
    get = (request: HttpRequest): HttpResponse -> {
      path = basePath + request.path
      type = getContentType(path)

      !type             => HttpResponse(status = 403)  # forbidden
      file.exists(path) => HttpResponse(body = file.readBin(path), headers = [ type
])
      _                 => HttpResponse(status = 404)  # not found
    }
  )
]

http.listen(8080, routes)
```

in a real application, `request.path` must be sanitized before being used to access files on the server; here, it is used directly only as an example

math

Mathematical operations and functions

Import

```
_ <- fat.math
```

Constants

- e, natural logarithm constant 2.71...
- maxInt, 9007199254740992
- minInt, -9007199254740992
- pi, ratio of circle to its diameter 3.14...

read more about number precision in FatScript

Basic functions

Name	Signature	Brief
abs	(x: Number): Number	Return absolute value of x
ceil	(x: Number): Number	Return smallest integer >= x
floor	(x: Number): Number	Return largest integer <= x
isInf	(x: Number): Boolean	Return true if x is infinity
isNaN	(x: Any): Boolean	Return true if x is not a number
logN	(x: Number, base: Number = e): Number	Return logarithm of x
random	(): Number	Return pseudo-random, where 0 <= n < 1
sqrt	(x: Number): Number	Return the square root of x
round	(x: Number): Number	Return the nearest integer to x

Trigonometric functions

Name	Signature	Brief
sin	(x: Number): Number	Return the sine of x
cos	(x: Number): Number	Return the cosine of x
tan	(x: Number): Number	Return the tangent of x
asin	(x: Number): Number	Return the arc sine of x
acos	(x: Number): Number	Return the arc cosine of x
atan	(x: Number, y = 1): Number	Return the arc tangent of x, y
radToDeg	(r: Number): Number	Convert radians to degrees
degToRad	(d: Number): Number	Convert degrees to radians

Hyperbolic functions

Name	Signature	Brief
sinh	(x: Number): Number	Return the hyperbolic sine of x
cosh	(x: Number): Number	Return the hyperbolic cosine of x
tanh	(x: Number): Number	Return the hyperbolic tangent of x

Statistical functions

Name	Signature	Brief
mean	(v: List/Number): Number	Return the mean of a vector
median	(v: List/Number): Number	Return the median of a vector

Name	Signature	Brief
sigma	(v: List/Number): Number	Return the standard deviation of a vector
variance	(v: List/Number): Number	Return the variance of a vector
max	(v: List/Number): Number	Return maximum value in vector
min	(v: List/Number): Number	Return the minimum value in vector
sum	(v: List/Number): Number	Return the sum of vector

Other functions

Name	Signature	Brief
fact	(x: Number): Number	Return the factorial of x
exp	(x: Number): Number	Return e raised to the power of x
sigmoid	(x: Number): Number	Return the sigmoid of x
relu	(x: Number): Number	Return the ReLU of x

Example

```
math <- fat.math   # named import
math.abs(-52)      # yields 52
```

See also

- Number (syntax)
- Number prototype extensions

recode

Data conversion between various formats

Import

`_ <- fat.recode`

> [type package](#) is automatically imported with this import

Constants

- numeric, regex definition used by `inferType` (DEPRECATED)

> the constant `numeric` is redundant and will be removed in version `3.x.x`

Variables

These settings can be adjusted to configure the behavior of the processing functions:

- csvSeparator, default is `,` (comma)
- csvReplacement, default is empty (just removes commas from text)
- xmlWarnings, default is `true` - set to `false` to suppress XML warnings (DEPRECATED)

Base64 functions

Name	Signature	Brief
toBase64	(data: Chunk): Text	Encode binary chunk to base64 text
fromBase64	(b64: Text): Chunk	Decode base64 text to original format

JSON functions

Name	Signature	Brief
toJSON	(_: Any): Text	Encode JSON from native types
fromJSON	(json: Text): Any	Decode JSON to native types

URL functions

Name	Signature	Brief
toURL	(text: Text): Text	Encode text to URL escaped text
fromURL	(url: Text): Text	Decode URL escaped text to original format
toFormData	(data: Scope): Text	Encode URL encoded Form Data from scope
fromFormData	(data: Text): Scope	Decode URL encoded Form Data to scope

CSV functions

Name	Signature	Brief
toCSV	(header: List/Text, rows: List/Scope): Text	Encode CSV from rows
fromCSV	(csv: Text): List/Scope	Decode CSV into rows

> `csvReplacement` is used by `toCSV` as replacement in case a `csvSeparator` is found within a text being encoded

XML functions (DEPRECATED)

XML attributes and self-closing tags are not supported.

Name	Signature	Brief

Name	Signature	Brief
toXML	(node: Any): Text	Encode XML from native types
fromXML	(text: Text): Any	Decode XML into native types

XML functions will be removed from FatScript standard libraries in version `3.x.x`, use XMLoaf

RLE functions

Name	Signature	Brief
toRLE	(chunk: Chunk): Chunk	Compress to RLE schema
fromRLE	(chunk: Chunk): Chunk	Decompress from RLE schema

Other functions

Name	Signature	Brief
inferType	(val: Text): Any	Convert text to void/boolean/number
minify	(src: Text): Text	Minifies FatScript source code

`minify` will replace any `$break` statements (debugger breakpoint) with `()`

Usage

JSON

Since FatScript alternatively accepts JSON-like syntax, `fromJSON` actually uses FatScript internal parser, which is blazing fast, but may or not yield exactly what one is expecting from a JSON parser.

For example, once the bellow fragment is parsed, since `null` in FatScript is absence of value, there would be no entry declarations for "prop":

```
"prop": null
```

Therefore, reading with `fromJSON` and writing back with `toJSON` is not necessarily an idempotent operation.

See also

- Type package
- SDK library

sdk

Fry's software development kit utilities

> a special library that exposes some of the inner elements of fry interpreter

Import

```
_ <- fat.sdk
```

Methods

Name	Signature	Brief
ast	(_): Void	Print abstract syntax tree of node
stringify	(_): Text	Converts node to json text
eval	(_): Any	Interprets text as FatScript program
getVersion	(): Text	Return fry version
printStack	(depth: Number): Void	Print execution context stack trace
readLib	(ref: Text): Text	Return fry library source code
typeOf	(_): Text	Return type name of node
getTypes	(): List	Return info about declared types
getDef	(name: Text): Any	Return type definition by name
getMeta	(): Scope	Return interpreter's metadata
setKey	(key: Text): Void	Set key for obfuscated bundles
setMem	(n: Number): Void	Set memory limit (node count)
runGC	(): Number	Run GC, return elapsed in milliseconds
quickGC	(): Number	Run single GC iteration and return ms
setAutoGC	(n: Number): Void	Set GC to run every n new nodes

Usage notes

readLib

```
_ <- fat.sdk
_ <- fat.console

print(readLib('fat.extra.Date'))  # prints the Date library implementation
```

> `readLib` cannot see external files, but `read` from file lib can

setKey

Use preferably on `.fryrc` file like so:

```
_ <- fat.sdk
setKey('secret')  # will encode and decode bundles with this key
```

See more about obfuscating.

setMem

Use preferably on `.fryrc` file like so:

```
_ <- fat.sdk
setMem(5000)  # ~2mb
```

Choosing between full, quick and auto GC

Most simple scripts in FatScript won't need to worry about memory management, as the default settings are designed to provide developers with a reasonably large memory capacity and sensible automatic behavior right out of the box.

The `quickGC` method offers swift, less exhaustive cleanup, making it suitable for scenarios where some leeway in memory allocation is acceptable. On the other hand, `runGC` ensures deterministic and thorough garbage collection, albeit at the expense of longer execution times, depending on various factors such as the size and complexity of the memory graph. However, in certain scenarios, `quickGC` may lead to a buildup of unreclaimed memory and might not be the most effective option.

In addition to manually choosing between `quickGC` and `runGC`, there is also a heuristic-based automatic GC. It is disabled by default, but can be enabled by calling `setAutoGC` with a non-zero value, this heuristic applies quickGC when ample free memory is available, ensuring minimal disruption. In contrast, under high memory pressure, fullGC is executed for comprehensive cleanup. This strategy balances memory efficiency with application performance, dynamically adapting to the memory usage pattern.

See more about memory management.

system

System-level operations and information

Import

```
_ <- fat.system
```

Types

Name	Signature	Brief
CommandResult	(code: ExitCode, out: Text)	Return type of capture

Constants

- successCode, 0: ExitCode
- failureCode, 1: ExitCode

Methods

Name	Signature	Brief
args	(): List/Text	Return list of args passed from shell
exit	(code: Number): *	Exit program with provided exit code
getEnv	(var: Text): Text	Get env variable value by name
shell	(cmd: Text): ExitCode	Execute cmd in shell, return exit code
capture	(cmd: Text): CommandResult	Capture the output of cmd execution
fork	(args: List/Text, out: Text = ø)	Start background process, return PID
kill	(pid: Number): Void	Send SIGTERM to process by PID
getLocale	(): Text	Get current locale setting
setLocale	(cmd: Text): Number	Set current locale setting
getMacId	(): Text	Get machine identifier (MAC address)
blockSig	(enabled: Boolean): Void	Block SIGINT, SIGHUP and SIGTERM

Usage Notes

Heads Up!

It is important to exercise caution and responsibility when using the `getEnv`, `shell`, `capture`, `fork` and `kill` methods. The `system` library provides the capability to execute commands directly from the operating system, which can introduce security risks if not used carefully.

To mitigate potential vulnerabilities, avoid using user input directly in constructing commands passed to these methods. User input should be validated to prevent command injection attacks and other security breaches.

Other Limitations (multithreading)

While the methods in this library support a variety of programming tasks, they are not optimized for interleaved usage within asynchronous [Workers]. When initiating processes from within threads, opt for `shell/capture` methods, or exclusively use `fork/kill`. Mixing these two method groups in multithreaded applications can result in unpredictable behavior.

> on each call, `shell/capture` will set `SIGCHLD` to its default behavior, while `fork` will ignore this signal to try to avoid zombie processes

get/set locale

The `fry` interpreter will attempt to initialize `LC_ALL` locale to `C.UTF-8` and if that locale is not available on the system tries to use `en_US.UTF-8`, otherwise, the default locale will be used.

See more about locale names.

locale configuration applies only to application, and is not persisted after `fry` exits

time

Time and date manipulation

Import

```
_ <- fat.time
```

number type is automatically imported with this import

Methods

Name	Signature	Brief
setZone	(offset: Number): Void	Set timezone in milliseconds
getZone	(): Number	Get current timezone offset
now	(): Epoch	Get current UTC in Epoch
format	(date: Text, fmt: Text = ø): Epoch	Convert Epoch to date format
parse	(date: Text, fmt: Text = ø): Epoch	Parse date to Epoch
wait	(ms: Number): Void	Wait for milliseconds (sleep)
getElapsed	(since: Epoch): Text	Return elapsed time as text

Usage Notes

Epoch

In FatScript time is represented as an arithmetic type so that you can do maths.

You can get the elapsed time between `time1` and `time2` like:

```
elapsed = time2 - time1
```

You can also check if `time2` happens after `time1`, simply like:

```
time2 > time1
```

format

Formats text date as "%Y-%m-%d %H:%M:%S.milliseconds" (default), when `fmt` is omitted.

milliseconds can only be transformed in default format, otherwise the precision is up to seconds

fmt parameter

The format specification is a text containing a special character sequence called conversion specifications, each of which is introduced by a '%' character and terminated by some other character known as a conversion specifier. All other characters are treated as ordinary text.

Specifier	Meaning
%a	Abbreviated weekday name
%A	Full weekday name
%b	Abbreviated month name
%B	Full month name
%c	Date/Time in the format of the locale
%C	Century number [00-99], the year divided by 100 and truncated to an integer
%d	Day of the month [01-31]
%D	Date Format, same as %m/%d/%y
%e	Same as %d, except single digit is preceded by a space [1-31]
%g	2 digit year portion of ISO week date [00,99]

Specifier	Meaning
%F	ISO Date Format, same as %Y-%m-%d
%G	4 digit year portion of ISO week date
%h	Same as %b
%H	Hour in 24-hour format [00-23]
%I	Hour in 12-hour format [01-12]
%j	Day of the year [001-366]
%m	Month [01-12]
%M	Minute [00-59]
%n	Newline character
%p	AM or PM string
%r	Time in AM/PM format of the locale
%R	24-hour time format without seconds, same as %H:%M
%S	Second [00-61], the range for seconds allows for a leap second and a double leap second
%t	Tab character
%T	24-hour time format with seconds, same as %H:%M:%S
%u	Weekday [1,7], Monday is 1 and Sunday is 7
%U	Week number of the year [00-53], Sunday is the first day of the week
%V	ISO week number of the year [01-53]. Monday is the first day of the week. If the week containing January 1st has four or more days in the new year then it is considered week 1. Otherwise, it is the last week of the previous year, and the next year is week 1 of the new year.
%w	Weekday [0,6], Sunday is 0
%W	Week number of the year [00-53], Monday is the first day of the week
%x	Date in the format of the locale
%X	Time in the format of the locale
%y	2 digit year [00,99]
%Y	4-digit year (can be negative)
%z	UTC offset string with format +HHMM or -HHMM
%Z	Time zone name
%%	% character

Under the hood `format` uses C's strftime and `parse` uses C's strptime, but the above format specification table applies pretty much both ways.

type._

Prototype extensions for native types:

- Void
- Boolean
- Number
- HugeInt
- Text
- Method
- List
- Scope
- Error
- Chunk

> FatScript **does not** load these definitions automatically into global scope, therefore you have to **explicitly** import those where needed

Importing

If you want to make all of them available at once you can simply write:

```
_ <- fat.type._
```

...or import one-by-one, as needed, e.g.:

```
_ <- fat.type.List
```

Common trait

All types on this package support the following prototype methods:

- apply (constructor)
- isEmpty
- nonEmpty
- size
- toText

See also

- Types (syntax)

Void

Void prototype extensions

Import

```
_ <- fat.type.Void
```

Constructor

Name	Signature	Brief
Void	(val: Any)	Return null, just ignore argument

Prototype members

Name	Signature	Brief
isEmpty	(): Boolean	Return true, always
nonEmpty	(): Boolean	Return false, always
size	(): Number	Return 0, always
toText	(): Text	Return 'null' as text

Example

```
_ <- fat.type.Void
x.isEmpty  # true, since x has not been declared
```

See also

- Void (syntax)
- Type package

Boolean

Boolean prototype extensions

Import

```
_ <- fat.type.Boolean
```

Constructor

Name	Signature	Brief
Boolean	(val: Any)	Coerces value to boolean

Prototype members

Name	Signature	Brief
isEmpty	(): Boolean	Return true if false
nonEmpty	(): Boolean	Return false if true
size	(): Number	Return 1 if true, 0 if false
toText	(): Text	Return 'true' or 'false' as text

Examples

```
_ <- fat.type.Boolean

x = true
x.isEmpty  # false, since x is true

Boolean('false')  # yields true, because text is non-empty
Boolean('')       # yields false, because text is empty
```

> note that the constructor does not attempt to convert value from text, which is consistent with flow control evaluations, and you can use a simple case if you need to make conversion from text to boolean

See also

- Boolean (syntax)
- Type package

Number

Number prototype extensions

Import

```
_ <- fat.type.Number
```

Aliases

- Epoch: unix epoch time in milliseconds
- ExitCode: exit status or return code
- Millis: duration in milliseconds

Constructor

Name	Signature	Brief
Number	(val: Any)	Text to number or collection size

performs the conversion from text to number assuming decimal base

Prototype members

Name	Signature	Brief
isEmpty	(): Boolean	Return true if zero
nonEmpty	(): Boolean	Return true if non-zero
size	(): Number	Return absolute value, same as math.abs
toText	(): Text	Return number as text
format	(fmt: Text): Text	Return number as formatted text
truncate	(): Number	Return number discarding decimals

Example

```
_ <- fat.type.Number
x = Number('52')  # number: 52
x.toText          # text: '52'
x.format('.2')    # text: '52.00'
```

format

The `format` method is used to convert numbers into strings in various ways. The basic structure of a format specifier is `%[flags][width][.precision][type]`. Here's what each of these components mean:

- `flags` are optional characters that control specific formatting behavior. For example, `0` can be used for zero-padding and `-` for left-justification.

- `width` is an integer that specifies the minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces or zeros, depending on the flag used.

- `precision` is an optional number following a `.` that specifies the number of digits to be printed after the decimal point.

- `type` is a character that specifies how the number should be represented. The common types are `f` (fixed-point notation), `e` (exponential notation), `g` (either fixed or exponential depending on the magnitude of the number), and `a` (hexadecimal floating-point notation).

Examples:

- `%5.f`: This will print the number with a total width of 5 characters, with no digits after the decimal point (because the precision is `f`, which means fixed-point, but no number follows the dot). It will be right-justified because no `-` flag is used.

- `%05.f`: Similar to the above, but because the `0` flag is used, the empty spaces will be filled with zeros.
- `%8.2f`: This will print the number with a total width of 8 characters, with 2 digits after the decimal point.
- `%-8.2f`: Similar to the above, but the number will be left-justified because of the `-` flag.
- `%.2e`: This will print the number using exponential notation, with 2 digits after the decimal point.
- `%.2a`: This will print the number using hexadecimal floating-point notation, with 2 digits after the hexadecimal point.
- `%.2g`: This will print the number in either fixed-point or exponential notation, depending on its magnitude, with a maximum of 2 significant digits.

if the `%` symbol is not present, `fmt` is automatically evaluated as `%<fmt>f`

See also

- Number (syntax)
- Math library
- Type package

HugeInt

HugeInt prototype extensions

Import

```
_ <- fat.type.HugeInt
```

Constructor

Name	Signature	Brief
HugeInt	(val: Any)	Number or text parsing to HugInt

> performs the conversion from text to number assuming hexadecimal representation

Prototype members

Name	Signature	Brief
isEmpty	(): Boolean	Return true if zero
nonEmpty	(): Boolean	Return true if non-zero
size	(): Number	Return number of bits needed to represent
toText	(): Text	Return number as hexadecimal text
modExp	(exp: HugeInt, mod: HugeInt): HugeInt	Return modular exponentiation
toNumber	(): Number	Converts to number (with precision loss)

Usage notes

When converting from `Number` type to `HugeInt`, the limit is 2^53, which is the maximum value that can be safely represented as an integer without precision loss. Attempting to pass a value higher than this limit will raise a `ValueError`.

Conversely, when converting from `HugeInt` to `Number`, values up to 2^1023 - 1 can be converted with some degree of precision loss. Attempting to convert a value higher than this will result in `infinity`, which can be checked using the `isInf` method provided by the [math library](#).

> the math library also provides the `maxInt` value, which serves to assess potential precision loss; if a number is less than `maxInt`, its conversion from `HugeInt` is considered safe without precision loss

See also

- [HugeInt (syntax)](#)
- [Type package](#)

Text

Text prototype extensions

Import

```
_ <- fat.type.Text
```

Constructor

Name	Signature	Brief
Text	(val: Any)	Coerces value to text, same as .toText

Prototype members

Name	Signature	Brief
isEmpty	(): Boolean	Return true if length is zero
nonEmpty	(): Boolean	Return true if non-zero length
size	(): Number	Return text length
toText	(): Text	Return self value
replace	(old: Text, new: Text): Text	Replace old with new (all)
indexOf	(frag: Text): Number	Get fragment index, -1 if absent
contains	(frag: Text): Boolean	Check if text contains fragment
count	(frag: Text): Number	Get repetition count for fragment
startsWith	(frag: Text): Boolean	Check if starts with fragment
endsWith	(frag: Text): Boolean	Check if ends with fragment
split	(sep: Text): List/Text	Split text by sep into list
toLower	(): Text	Return lowercase version of text
toUpper	(): Text	Return uppercase version of text
trim	(): Text	Return trimmed version of text
match	(regex: Text): Boolean	Return text is match for regex
repeat	(n: Number): Text	Return text repeated n times
overlay	(base: Text, align: Text): Text	Return text overlaid on base
patch	(i, n, val: Text): Text	Inserts val at i, removing n chars

Example

```
_ <- fat.type.Text
x = 'banana'
x.size                       # yields 6
x.replace('nana', 'nquet');  # yields 'banquet'
```

Regex

When defining regular expressions, prefer to use [raw texts]() and remember to escape backslashes as needed, ensuring that the regular expressions are interpreted correctly.

At the moment, regex support is limited to matching only:

```
alphaOnly = "^[[:alpha:]]+$"
'abc'.match(alphaOnly)  # output: true
```

the implemented dialect is [POSIX regex extended]()

Overlay

The default align value (if not provided) is 'left'. Other possible values are 'center' and 'right':

```
'x'.overlay('___')            # 'x__'
'x'.overlay('___', 'left')    # 'x__'
'x'.overlay('___', 'center')  # '_x_'
'x'.overlay('___', 'right')   # '__x'
```

the outcome is always the same size as `base` parameter, the text will be cut if it is longer

See also

- Text (syntax)
- Type package

Method

Method prototype extensions

Import

```
_ <- fat.type.Method
```

Constructor

Name	Signature	Brief
Method	(val: Any)	Wrap val in a method

Prototype members

Name	Signature	Brief
isEmpty	(): Boolean	Return false, always
nonEmpty	(): Boolean	Return true, always
size	(): Number	Return 1, always
toText	(): Text	Return 'Method' text literal
arity	(): Number	Return method arity

Example

```
_ <- fat.type.Method
x = (): Number -> 3
(~ x).toText  # yields 'Method'
```

> note that it is necessary to explicitly opt out of using automatic calls to make use of the prototype members

See also

- Method (syntax)
- Type package

List

List prototype extensions

Import

```
_ <- fat.type.List
```

Constructor

Name	Signature	Brief
List	(val: Any)	Wrap val into a list

Prototype members

Name	Signature	Brief
isEmpty	(): Boolean	Returns true if length is zero
nonEmpty	(): Boolean	Returns true if length is non-zero
size	(): Number	Returns list length
toText	(): Text	Returns 'List' as text literal
join	(sep: Text): Text	Joins list with separator into text
flatten	(): List	Flattens list of lists into one list
find	(p: Method): Any	Returns first matching item or null
contains	(p: Method): Boolean	Checks if list contains match for predicate
filter	(p: Method): List	Returns sub-list matching predicate
reverse	(): List	Returns a reversed copy of the list
shuffle	(): List	Returns a shuffled copy of the list
unique	(): List	Returns a list of unique items
sort	(): List	Returns a sorted copy of the list
sortBy	(key: Any): List	Returns a sorted copy of the list *
indexOf	(item: Any): Number	Returns item index, -1 if absent
head	(): Any	Returns first item, null if empty
tail	(): List	Returns all items, but the first
map	(m: Method): List	Functional utility (allows chaining)
reduce	(m: Method, acc: Any): Any	Functional utility
walk	(m: Method): Void	Apply side-effects to each item
patch	(i, n, val: List): List	Inserts val at position i, removing n items
headOption	(): Option	Returns first item, as Option
itemOption	(index: Number): Option	Get item by index, as Option
findOption	(p: Method): Option	Search item by predicate, as Option

Example

```
_ <- fat.type.List
x = [ 'a', 'b', 'c' ]
x.size  # yields 3
```

Sorting

The `sort` and `sortBy` methods implement the quicksort algorithm, enhanced with random pivot selection. This approach is known for its efficiency, offering an average-case time complexity of O(n log n). It demonstrates high performance across most datasets. For datasets containing duplicate values or keys, stable sorting cannot be guaranteed, and performance may degrade to O(n^2) in the worst case, where all elements are identical or have the same key.

`sortBy` accepts a textual parameter for `key` if it is a list of `Scope`, or a numerical parameter if it is a list of `List` (matrix), representing the index

Reducing

The `reduce` method in FatScript transforms a list into a single value by applying a reducer (`m: Method`) to each element in sequence, starting from an initial accumulator value (`acc: Any`), or from the first element if no value is provided. This method is useful for operations that involve aggregating data from a list.

Characteristics

- **Reducer Method:** The reducer should take the current accumulator value and the current list item, returning the updated accumulator value.
- **Empty List Behavior:** When `reduce` is applied to an empty list without an initial accumulator value, it returns `null`.

Practical Example

```
_ <- fat.type.List
sumReducer = (acc: Number, item: Number) -> acc + item
sum = [1, 2, 3].reduce(sumReducer)  # yields 6
```

for complex data transformations or when dealing with lists of scopes, carefully structure the reducer to handle the specific data types and desired output

See Also

- [List (syntax)]
- [Option type]
- [Type package]

Scope

Scope prototype extensions

Import

```
_ <- fat.type.Scope
```

Constructor

Name	Signature	Brief
Scope	(val: Any)	Wrap val into a scope

Prototype members

Name	Signature	Brief
isEmpty	(): Boolean	Return true if size is zero
nonEmpty	(): Boolean	Return true if non-zero size
size	(): Number	Return number of entries
toText	(): Text	Return 'Scope' text literal
copy	(): Scope	Return deep copy of scope
keys	(): List	Return list of scope keys
maybe	(key: Text): Option	Return Option wrapped value

Example

```
_ <- fat.type.Scope
x = { num = 12, prop = 'other' }
x.size  # yields 2
```

See also

- Scope (syntax)
- Option type
- Type package

Error

Error prototype extensions

Import

```
_ <- fat.type.Error
```

Aliases

- AssignError: assigning a new value to an immutable entry
- AsyncError: asynchronous operation failure
- CallError: a call is made with insufficient arguments
- FileError: file operation failure
- IndexError: index is out of list/text bounds
- KeyError: the key (name) is not found in scope
- SyntaxError: syntax or code structure error
- TypeError: type mismatch on method call, return, or assign
- ValueError: type may be okay, but content is not accepted

Constructor

Name	Signature	Brief
Error	(val: Any)	Return val coerced to text wrapped in error

Prototype members

Name	Signature	Brief
isEmpty	(): Boolean	Return true, always
nonEmpty	(): Boolean	Return false, always
size	(): Number	Return 0, always
toText	(): Text	Return error text val

Example

```
_ <- fat.type.Error
x = Error('ops')
x.toText  # yields "Error: ops"

# ...or something unexpected
e = undeclared.item  # raises Error
e.toText             # yields "can't resolve scope of 'item'"
```

See also

- Failure library
- Error (syntax)
- Type package

Chunk

Chunk prototype extensions

Import

```
_ <- fat.type.Chunk
```

Constructor

Name	Signature	Brief
Chunk	(val: Any)	Coerces value to chunk (binary)

Prototype members

Name	Signature	Brief
isEmpty	(): Boolean	Returns true if size is zero
nonEmpty	(): Boolean	Returns true if non-zero size
size	(): Number	Returns chunk size (in bytes)
toText	(): Text	Converts chunk to text format
toBytes	(): List/Number	Converts chunk to bytes list
seekByte	(byte: Number, offset: Number = 0): Number	Returns index of first match
patch	(i, n, val: Text): Text	Inserts val at i, removing n bytes

`toText` replaces any invalid UTF-8 sequences with U+FFFD, represented as � in UTF-8

Example

```
_ <- fat.type.Chunk

x = Chunk('example')

x.size     # yields 7
x.toText   # yields 'example'
x.toBytes  # yields [ 101, 120, 97, 109, 112, 108, 101 ]
```

See also

- [Chunk (syntax)]
- [Type package]

extra._

Additional types implemented in vanilla FatScript:

- Date - Calendar and date handling
- Duration - Millisecond duration builder
- HashMap - Quick key-value store
- Logger - Logging support
- Memo - Generic memoization utility
- Option - Encapsulation of optional value
- Param - Parameter presence and type verification
- Sound - Sound playback interface
- Storable - Data store facilities

Importing

If you want to make all of them available at once you can simply write:

```
_ <- fat.extra._
```

...or import one-by-one, as needed, e.g.:

```
_ <- fat.Date
```

Developer note

Currently most of these utilities are not resource or performance optimized.

The intent here was more of providing simple features, as basic templates that can be pulled out via readLib, so any developer with particular requirements will have a starting point for their own implementations.

Date

Calendar and date handling

> operations like addition and subtraction of days, months, and years, ensuring accurate handling of various date-related complexities such as leap years and month-end calculations

Import

```
_ <- fat.extra.Date
```

> time library, math library, Error type, Text type, List type, Number type, Duration type are automatically imported with this import

Date Type

`Date` offers a comprehensive solution for managing dates, including leap years and time of day.

Properties

- `year`: Number - Year of the date
- `month`: Number - Month of the date
- `day`: Number - Day of the date
- `tms`: Millis - Time of the day in milliseconds

> default value points to: 1 of January of 1970

Prototype Members

Name	Signature	Brief
fromEpoch	(ems: Epoch): Date	Creates a Date instance from an epoch time
isLeapYear	(year: Number): Boolean	Determines if a year is a leap year
normalizeMonth	(month: Number): Number	Normalizes the month number
daysInMonth	(year: Number, month: Number): Number	Returns number of days in month of year
isValid	(year, month, day, tms): Boolean	Validates the date components
truncate	(): Date	Truncates the time of day
toEpoch	(): Epoch	Converts the Date instance to epoch time
addYears	(yearsToAdd: Number): Date	Adds years to the date
addMonths	(monthsToAdd: Number): Date	Adds months to the date
addWeeks	(weeksToAdd: Number): Date	Adds weeks to the date
addDays	(daysToAdd: Number): Date	Adds days to the date

Usage Examples

```
_ <- fat.extra.Date

# Create a Date instance
myDate = Date(2023, 1, 1)

# Add one year to the date
newDate = myDate.addYears(1)

# Add two weeks to a date
datePlusTwoWeeks = myDate.addWeeks(2)

# Create a Date from epoch time (in milliseconds)
# result is influenced by current timezone, see: time.setZone
epochTime = 1672531200000
dateFromEpoch = Date.fromEpoch(Epoch(epochTime))
```

```
# Convert a date to epoch time
epochFromDate = myDate.toEpoch
```

Duration

Millisecond duration builder

In FatScript time is natively expressed in milliseconds, and this type provides a simple way to express different time magnitudes effortlessly into Millis.

Import

```
_ <- fat.extra.Duration
```

Constructor

Name	Signature	Brief
Duration	(val: Number)	Create a Millis duration converter

Prototype members

Name	Signature	Brief
nanos	(): Millis	Interpret value as nanoseconds
micros	(): Millis	Interpret value as microseconds
millis	(): Millis	Interpret value as milliseconds
seconds	(): Millis	Interpret value as seconds
minutes	(): Millis	Interpret value as minutes
days	(): Millis	Interpret value as days
weeks	(): Millis	Interpret value as weeks
months	(): Millis	Interpret value as months (aprox.)
years	(): Millis	Interpret value as years (aprox.)

Example

```
_    <- fat.extra.Duration
time <- fat.time

fiveSeconds = Duration(5).seconds
time.wait(fiveSeconds)  # sleeps thread for 5 seconds
```

HashMap

An optimized in-memory key-value store, serving as a better performance replacement for default Scope implementation, designed for handling large data sets efficiently.

the speed gains will come at the expense of more memory usage

Import

```
_ <- fat.extra.HashMap
```

Constructor

Name	Signature	Brief
HashMap	(capacity: Number = 97)	Create a HashMap with a specified capacity

the default capacity of 97 is generally efficient for up to 10,000 items

Capacity Optimization

Ideally, you should keep at most about 100 items per 'bucket' in the hash table. In this context, 'capacity' refers to the number of buckets available for your data. Note that this implementation does not automatically adjust its size, so proper initial sizing is crucial. The following table can help determine the optimal capacity for storing `n` items:

```
n <   5000 => 53
n <  10000 => 97
n <  20000 => 193
n <  40000 => 389
n <  80000 => 769
n < 160000 => 1543
_          => 3079
```

using prime numbers can help reduce collisions

These values are based on empirical tests and should be adjusted according to your specific data needs and performance goals. Keep in mind that the relationship between capacity and performance is not entirely linear; as the number of items increases, the benefits of further increasing the capacity diminish.

Recommendation

Although the standard FatScript Scope exhibits slower performance for insertions and is particularly slow for deletions (such as setting to `null`), it excel in data retrieval and updates, outperforming `HashMap` for small collections (under ~500 items). Therefore, the benefits of using `HashMap` are most noticeable in scenarios involving frequent inserts and deletions on large data sets.

Prototype members

Name	Signature	Brief
isEmpty	(): Boolean	Returns true if length is zero
nonEmpty	(): Boolean	Returns true if length is non-zero
size	(): Number	Returns hash table length
toText	(): Text	Returns 'HashMap/capacity' as text literal
set	(key: Text, value: Any): Any	Set a key-value pair in the HashMap
get	(key: Text): Any	Get the value associated with a key
keys	(): List/Text	Return a list of all keys in the HashMap

Example

```
_ <- fat.extra.HashMap

hmap = HashMap()
```

```
hmap.set('key1', 'value1')

hmap.get('key1')  # yields 'value1'
hmap.keys         # yields [ 'key1' ]
```

Logger

Logging support

> from simple console logging to file-based logging

Import

```
_ <- fat.extra.Logger
```

> [console library](), [color library](), [file library](), [time library](), [sdk library](), and [type library]() are automatically imported with this import

Logger Type

`Logger` provides customizable logging capabilities with various levels and formats.

Properties

- `level`: Text (default 'debug') - Logging level
- `showTime`: Boolean (default true) - Flag to display timestamps

> valid levels: 'debug', 'info', 'warn', 'error'

Prototype members

Name	Signature	Brief
setLevel	(level: Text)	Sets the logging level
setShowTime	(showTime: Boolean)	Toggles timestamp display in logs
asMessage	(level: Text, args: Scope): Text	Formats log messages (can be overridden)
log	(msg: Any, fg: Number)	Outputs messages (can be overridden)

Logging methods

- `debug(_1, _2, _3, _4, _5)`: Logs a debug message
- `info(_1, _2, _3, _4, _5)`: Logs an info message
- `warn(_1, _2, _3, _4, _5)`: Logs a warning message
- `error(_1, _2, _3, _4, _5)`: Logs an error message

Subtypes

BoringLogger

- Inherits from `Logger`
- Overrides `log` to output plain text without color

FileLogger

- Inherits from `Logger`
- Additional Properties:
  - `logfile`: Text (default 'log.txt') - file for logging
- Overrides `log` to append messages to a file

Usage Example

```
_ <- fat.extra.Logger

# Create an instance with custom settings
myLogger = Logger(level = 'info', showTime = false)

# Log an information message
myLogger.info('This is an informational message.')
```

```
# Create a FileLogger to log messages to a file
fileLogger = FileLogger('myLog.txt')
fileLogger.info('Logged to file.')
```

Memo

Generic memoization utility (can also create lazy values)

Import

```
_ <- fat.extra.Memo
```

Constructor

Name	Signature	Brief
Memo	(method: Method)	Create a Memo instance for a method

> the arity of the memoized method should be 1 or else 0 (for lazy values)

Prototype members

Name	Signature	Brief
asMethod	(): Method	Return a memoized version of original method
call	(arg: Any): Any	Memoized call; cache and return results

Example

Memo is useful for optimizing functions by caching results. It stores the outcome of function calls and returns the cached result when the same inputs occur again.

```
_ <- fat.extra.Memo

fib = (n: Number) -> {
  n <= 2 => 1
  _      => quickFib(n - 1) + quickFib(n - 2)
}

quickFib = Memo(fib).asMethod

quickFib(50)  # 12586269025
```

You can now call `quickFib` as if you were calling `fib`, but with cached results for previously computed inputs.

> caveat: may cause memory allocation build-up

Option

Encapsulation of optional value

Import

```
_ <- fat.extra.Option
```

> [Error type](#) is automatically imported along with this import

Types

This library introduces two main constructs: `Some` and `None`, which are special cases of the `Option` type, providing a way to represent optional values, encapsulating the presence (`Some`) or absence (`None`) of a value.

Prototype members

Name	Signature	Brief
isEmpty	(): Boolean	Checks if the option is None
nonEmpty	(): Boolean	Checks if the option is Some
get	(): Any	Returns value or raises NoSuchElement
getOrElse	(default: Any): Any	Returns value or default if None
map	(fn: Method): Option	Applies method to contained value
flatMap	(fn: Method/Option): Option	Applies method that returns Option
filter	(predicate: Method): Option	Filters value by predicate
toList	(): List	Converts option to List
concrete	(): Option	Resolves option to Some or None

Usage Example

```
_ <- fat.extra.Option

# Creating options
x = Some(5)  # equivalent to Option(5).concrete
y = None()   # equivalent to Option().concrete

# Working with options
isEmptyX = x.isEmpty   # false
isEmptyY = y.isEmpty   # true
valX = x.getOrElse(0)  # 5
valY = y.getOrElse(0)  # 0

# Applying a transformation
transformedX = x.map(v -> v * 2).getOrElse(0)  # 10
transformedY = y.map(v -> v * 2).getOrElse(0)  # 0

# Lifting values to option
label: Text = Option(opVal).concrete >> {
  Some => 'some value'  # case where opVal is not null
  None => 'no value'    # case where opVal is null
}
```

Option in Functional Programming

In FatScript, `null` is integrated as a first-class citizen, enabling native types, in most cases, to handle absent values without necessitating additional constructs for safety. Consequently, the `Option` type is included in the `extra` package as a syntactic sugar.

It allows explicit encapsulation of optional values for semantic clarity or adherence to certain functional programming paradigms. An example of its utility is demonstrated in the `Scope` type, which includes a `maybe` method alongside the standard value retrieval syntax:

- `myScope('key')` returns the value associated with `key` or `null` if the key does not exist.
- `myScope.maybe('key')` provides an `Option` wrapped value, distinguishing explicitly between the existence (`Some`) and absence (`None`) of a value.

Semantic handling of missing values

One of the key benefits of using the `Option` type is its ability to handle operations with potentially missing values semantically and safely. This feature is particularly useful in primitive operations or data transformations where `null` values might otherwise lead to errors. For example, consider a scenario where you need to sum a number with a value that may not be present:

```
# Assuming eggsBought is defined and has a value
eggsBought: Number = ...

# fridge.maybe('egg') retrieves the number of eggs in the fridge as an Option
# If 'egg' is not present, it defaults to 0, avoiding null-related errors
totalEggs: Number = fridge.maybe('egg').getOrElse(0) + eggsBought
```

Performance considerations

The use of `Option` types introduces computational overhead due to function calls needed to manipulate values and additional memory stemming from their underlying structure. While the benefits of safety and expressiveness are significant, the performance cost could become noticeable in tight loops or when processing large datasets.

See also

- Scope type
- Error type

Param

Parameter presence and type verification

Import

```
_ <- fat.Param
```

Text type and Error type are automatically imported with this import

Types

This library introduces the `Param` type and the `Using` utility for implicit parameter declaration.

Constructors

Both `Param` and `Using` constructors take two arguments:

- **_exp**: the parameter name to check in context.
- **_typ**: the expected type of the evaluated value.

Param

The `Param` type provides mechanisms for checking the presence and type of parameters in the execution context.

Prototype members

Name	Signature	Brief
get	(): Any	Retrieves the parameter if it matches the type

the `get` method throws `KeyError` if the parameter is not defined, and `TypeError` if the type does not match

Example

```
_ <- fat.extra.Param

currentUser = Param('userId', 'Text')

...

# Assuming userId is defined in the context and is a text,
# safely retrieve it's value from the current namespace
userId = currentUser.get
```

Using

Apply `Using` to suppress implicit parameter hints on method declarations for entries expected to be in scope.

alternatively, to suppress warnings about implicit parameters, name the implicit entry starting with an underscore (_)

Example

```
_ <- fat.extra.Param

printUserIdFromContext = -> {
  Using('userId', 'Text')
  console.log(userId)
}
```

if the implicit parameter is missing or mismatched, an error will be raised at runtime when the method is called

See also

- Extra package

Sound

Sound playback interface

Wrapper for command-line audio players using fork and kill.

Import

```
_ <- fat.extra.Sound
```

Constructor

The `Sound` constructor takes three arguments:

- **path**: the filepath of your audio file.
- **duration** (optional): the cool off time (in milliseconds) to accept to play again the file, usually you want to set this to the exact duration of your audio.
- **player** (optional): the default player used is `aplay` (common Linux audio utility, only supports wav files), but you could use `ffplay` to play mp3, for example, defining `ffplay = [ 'ffplay', '-nodisp', '-autoexit', '-loglevel', 'quiet' ]`, then providing it as argument for your sound instance. In this case the package `ffmpeg` needs to be installed on the system.

Prototype members

Name	Signature	Brief
play	(): Void	Start player, if not already playing
stop	(): Void	Stop player, if still playing

> state of "still playing" is inferred from the duration parameter

Example

```
_    <- fat.extra.Sound
time <- fat.time

applause = Sound('applause.wav', 5000);
applause.play
time.wait(5000)
```

> note that Sound spawns a child process to play the audio, so it is asynchronous

Sound in Web Build

When using `fry` built with Emscripten (for example, when using FatScript Playground), this prototype uses embedded commands `$soundPlay` and `$soundStop`, which are only defined in the web build. Therefore, instead of utilizing a CLI audio player through process forking, there is audio support via SDL2/WebAudio.

See also

- Extra package

Storable

Data store facilities

Import

```
_ <- fat.extra.Storable
```

> file library, sdk library, enigma library, Error type, Text type, Void type and Method type are automatically imported with this import

Mixins

This library introduces two mixin types: `Storable` and `EncryptedStorable`

Storable

The `Storable` mixin provides methods for storing and retrieving objects in the filesystem using JSON serialization.

Prototype members

Name	Signature	Brief
list	(): List/Text	Gets list of ids for stored instances
load	(id: Text): Any	Loads an object from the filesystem
save	(): Boolean	Saves the current object instance
erase	(): Boolean	Deletes the file associated with the id

> the `load` and `save` methods throw `FileError` on failure

EncryptedStorable

Extends `Storable` with encryption capabilities for safer data storage. Requires an implementation of `getEncryptionKey` method.

Usage example

```
_ <- fat.extra.Storable

# Define a type that includes Storable (or EncryptedStorable)
User = (
  Storable  # Include the Storable mixin

  # EncryptedStorable                             # alternative implementation
  # getEncryptionKey = (): Text -> '3ncryp1ptM3'  # could get via KMS or config

  ## Argument slots
  name: Text
  email: Text

  # Setters return new immutable instance copy with updated field
  setName = (name: Text) -> self + User * { name }
  setEmail = (email: Text) -> self + User * { email }
)

# Create a new user instance
newUser = User('Jane Doe', 'jane.doe@example.com')

# Save the new user
newUser.save

# Update a user's information and save the changes
updatedUser = newUser
```

```
  .setName('Jane Smith')
  .setEmail('jane.smith@example.com')
updatedUser.save

# List all saved users
userIds = User.list

# Load a user from the filesystem
userId = userIds(0)  # ...or newUser.id
loadedUser = User.load(userId)

# Delete user's data from the filesystem
loadedUser.erase  # ...or User.erase(userId)
```

Storable in Web Build

When using `fry` built with Emscripten (for example, when using FatScript Playground), this prototype uses embedded commands `$storableSet`, `$storableGet`, `$storableList`, and `$storableRemove`, which are only defined in the web build. Therefore, instead of using the conventional file system for storage, there is special support for using the browser's `localStorage` object.

See also

- Extra package

Embedded commands

Embedded commands are FatScript's low-level functions that can be invoked with keywords preceded by a dollar sign `$`. These commands are always available, implemented as compiled code, and require no imports.

Unlike methods, they take no explicit arguments, but may read from specific entry names in the current scope, or even from the interpreter's internal state.

Handy ones

Here a are some embedded commands that could be useful to know:

- `$break` pauses execution and loads the debugging console
- `$debug` toggles interpreter debug logs
- `$exit` exits program with provided code
- `$keepDotFry` keeps the config (.fryrc) in scope after startup
- `$result` toggles result printing at the end of execution
- `$root` provides a reference to global scope
- `$self` provides a self reference to method/instance scope
- `$bytesUsage` returns total of bytes allocated at the moment
- `$nodesUsage` returns total of nodes allocated at the moment
- `$isMain` checks if code is executing as main or module

> `root` and `self` keywords are automatically lifted into `$root` and `$self`

You can call those directly on your code, like:

```
$exit  # terminates the program
```

> in order to use other embedded commands you have to study the C implementation of `fry`, as the complete list is not documented, refer to embedded.c file

Libs under the hood

Standard libraries wrap embedded calls into methods, providing a more ergonomic interface. You don't need to create an execution scope or load arguments into that scope before delegating execution to them.

For example, here's how you can use the `floor` method from math lib:

```
_ <- fat.math
floor(2.53)
```

This method is implemented as:

```
floor = (x: Number): Number -> $floor
```

Under the hood, the `floor` method creates an execution scope and loads an argument as `x` into it. The method then delegates execution to the `$floor` embedded command, which reads the value of `x` from the current scope and returns the floor of that number.

You can achieve the same outcome as above method by doing the following:

```
x = 2.53
$floor  # reads value of x from current scope
```

Hacking

You can see which embedded command a library method is calling by looking into the library's implementation via the `readLib` method from the [SDK lib](SDK lib). Technically, there is nothing preventing you from calling embedded commands directly.

For example, you could terminate your program by calling `$exit` directly, which will exit with code 0 (default) or, if a numeric entry named `code` exists in the current scope, the value of that entry will be used as the exit code. However, it would be more elegant to import the `fat.system` library and call the `exit` method with the desired exit code:

```
sys <- fat.system
sys.exit(0)  # exits with code 0
```

This approach makes your code more readable and less prone to errors, and it also provides a better separation of concerns.

It's important to keep in mind that embedded commands are black boxes and not intended for writing common FatScript code. In most cases, you would need to read the underlying C implementation to better grasp what a command is actually doing.

While it's possible to use embedded commands to gain additional runtime performance by avoiding imports and method calls, this is not recommended due to the loss of code readability. In general, it's better to use the standard libraries and follow best practices for writing clear, maintainable code.