the TSAK

Slightly longer introduction.

A bit of history, quick description and overview of current status.

First, a bit of history...

About a two years ago, I've conceive the idea of a tool that been named as "Telemetry Swiss Army Knife". And the name reflects the idea behind this tool: transformation and generation of random data originated from random sources to New Relic events, metrics and logs data. Imagine, something similar to AWK, but with better math, more ways for input data and geared toward telemetry output.

While the project shows a very promising signs to produce a very useful tool, the "old TSAK" didn't become what I hope to be. So, I froze the project. But the problems that to be solved with use of TSAK didn't go away in those two years. What was changes in this two years is my understanding of how this tool have to be implemented to be a more efficient and solve wider class of problems that still haunting us. Therefore ...

I’ve decided to resurrect the idea of programmatic data transformation from “something” to telemetry and TSAK as the tool standing behind it.

The new TSAK is a programmatic instrument designed to obtain and transform data of various format from variety of sources and convert this data to or generate a set of telemetry data that is submitted to New Relic platform. Rather than very rigid and curated instrument the old TSAK was, the new TSAK is just a programming language that is close to it’s “look and feel” to JavaScript or any other languages of that type, enhanced with specialized primitives that helping to process data and generate telemetry. In the next cards I’ll try to show a use cases for TSAK and how you can solve particular problems using this tool.

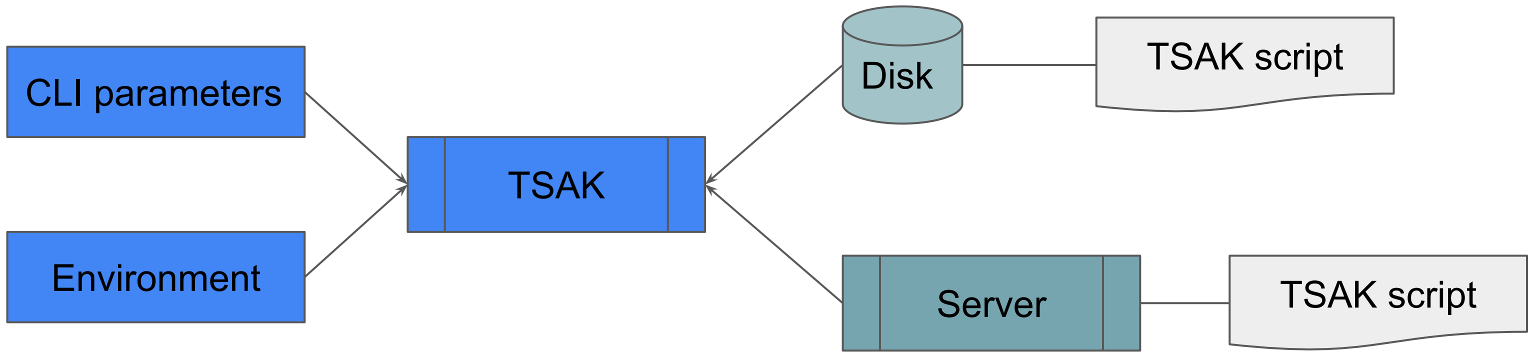
WARNING

At the time of writing, in January of 2023, this tool still in very early stages of development. Currently, there are some demonstrable features, that could be acting as POC to the cause. Every week, I am working to bring a new features, that enhance and improve ability of that tool. Author are always open for suggestions and critique.

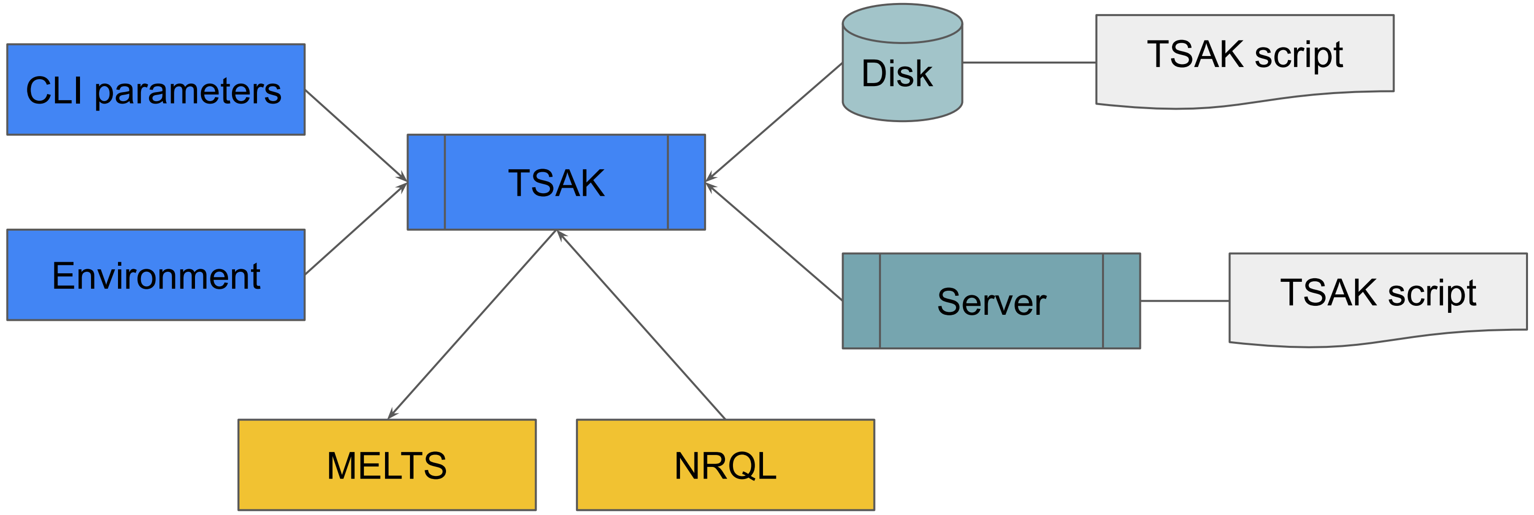
WARNING

Generic architecture of TSAK.

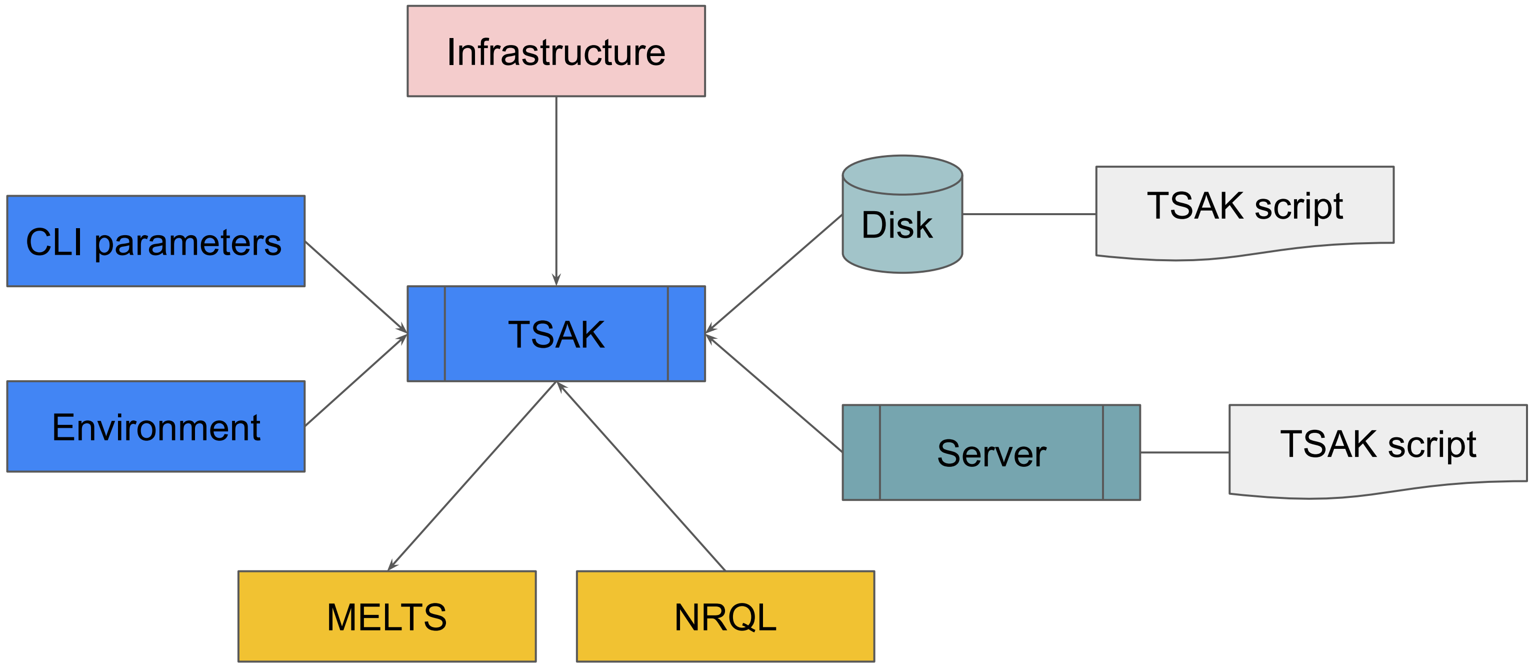
TSAK consists of a single, self-containing execution binary, compiled for targeted architecture and do not carry of any external dependencies. All TSAK standard library functionality is embedded inside the tool, that makes distribution and deployment simple. TSAK scripts can be deployed ether locally, or downloaded from ftp:// or http:// , https:// sources during execution. TSAK do not have a mandatory configuration file and all aspects of TSAK environment either controlled from TSAK command line parameters, or environment variables or through the scripts.



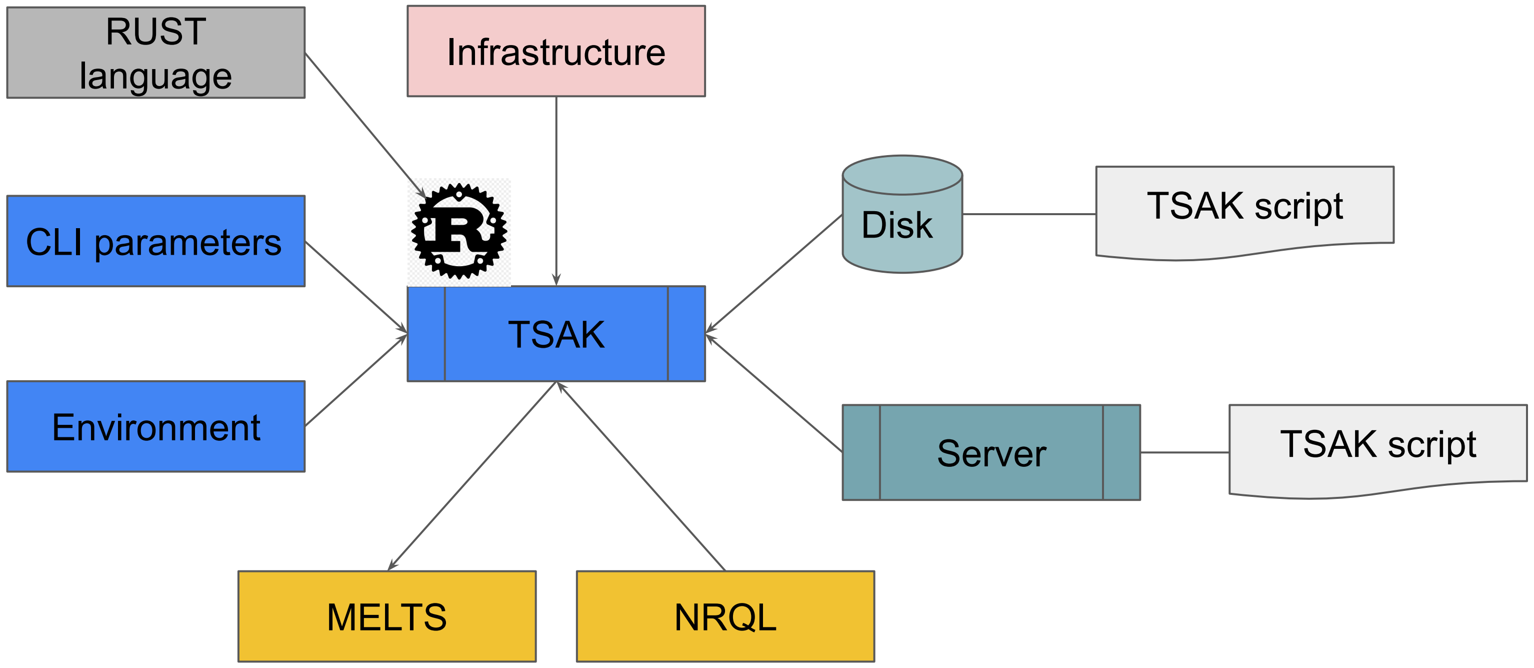
TSAK can interact with New Relic platform by ether sending Metrics/Events/Logs/Traces to New Relic REST API endpoints. Or can query New Relic NRDB by sending NRQL and programmatically process data received.



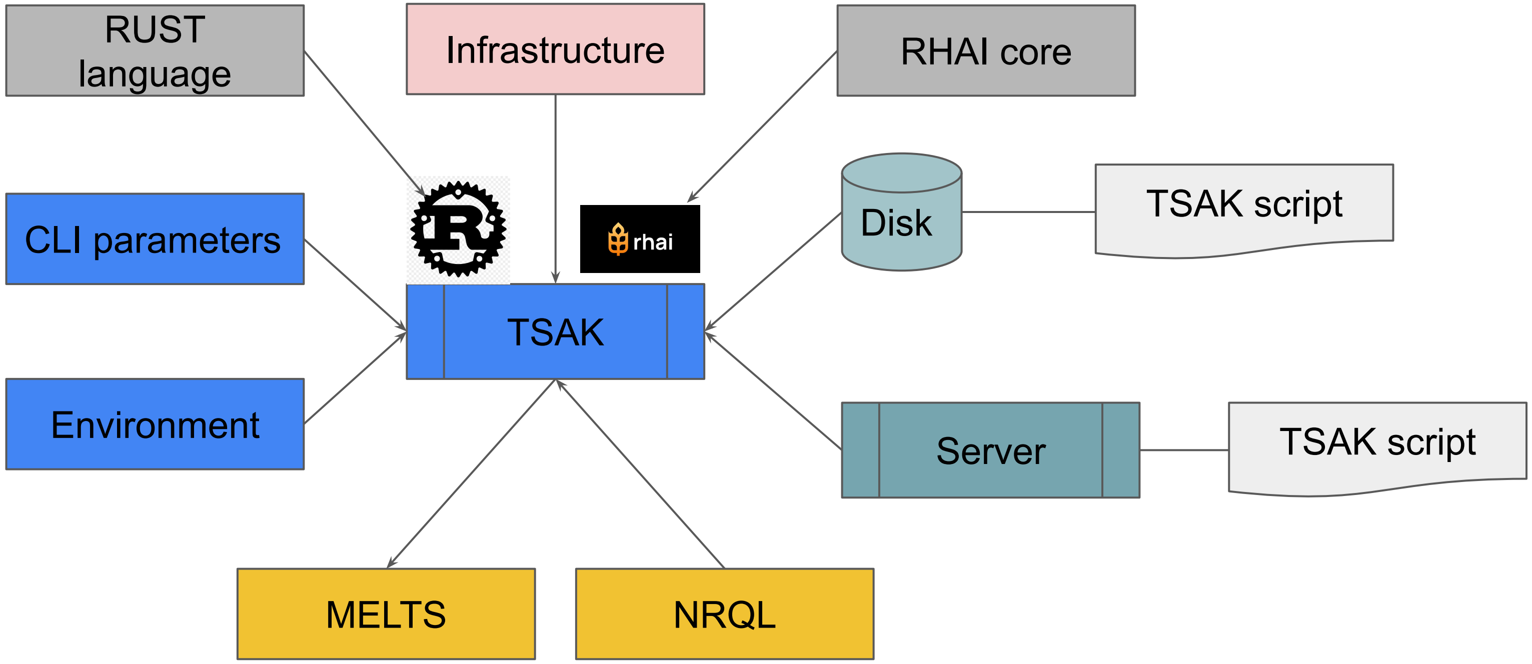
TSAK can also obtain certain information from local infrastructure (including localhost) by using TSAK standard library functions and capabilities.



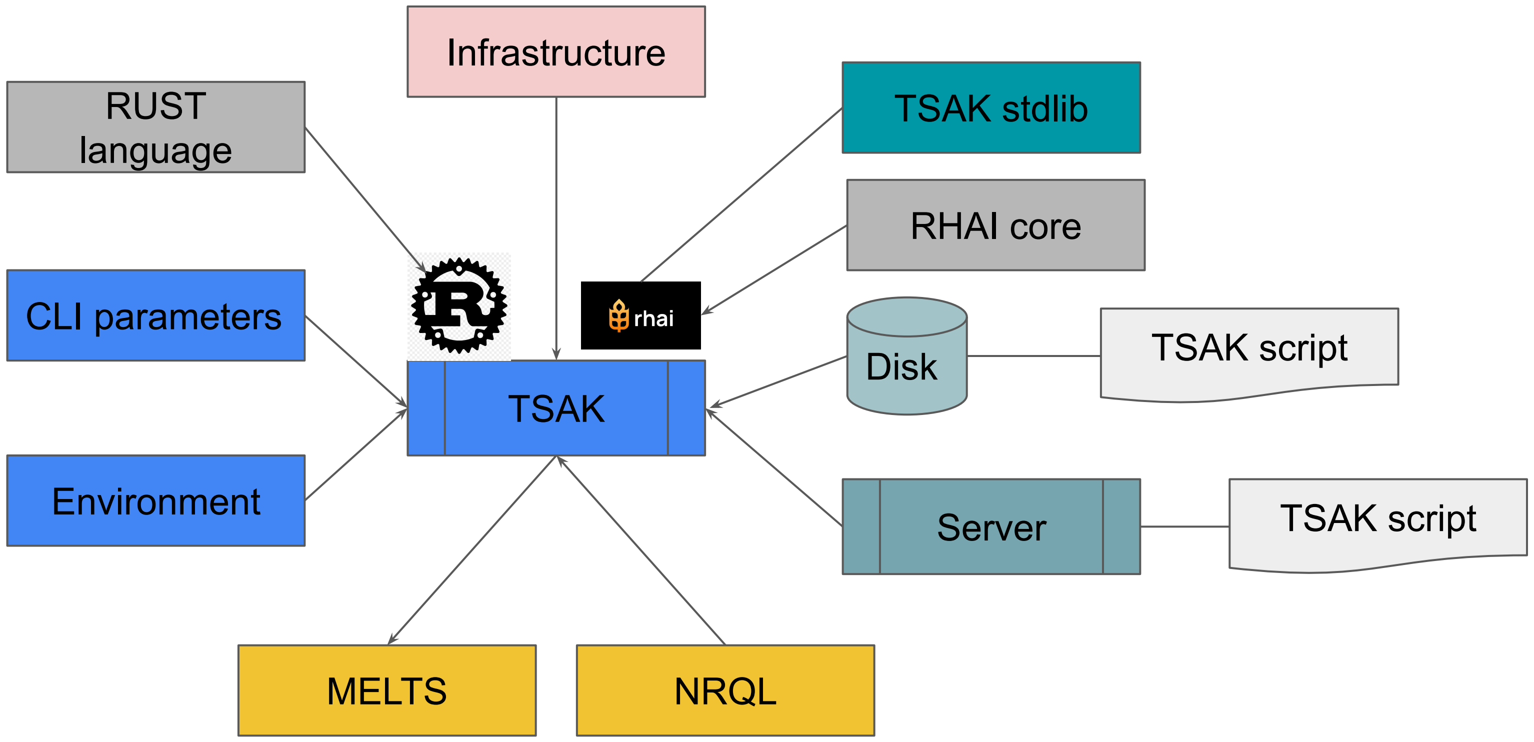
TSAK implemented using RUST language, which gives it a high performance, near equalled with provided by C as well as high memory and application safety standards surpassing what's offered by other programming platforms.



In the core of the TSAK scripting engine is a RUST module implementing RHAI scripting language engine. RHAI is embeddable, fast and efficient scripting core



In the core of the TSAK scripting engine is a RUST module implementing RHAI scripting language engine. RHAI is embeddable, fast and efficient scripting core



Scripting language.

Why RHAI ? Why not take something embedded Python, Ruby or any other scripting engines ?

When you are developing a solution that includes DSL (Domain Specific Language), choosing the language core that already exists and was developed as “generic” programming language wouldn’t be a primary choice due to the specifics of the tool that you are designing and developing. Here is just some of the requirements that exists for the scripting language core fort the TSAK tool:

- Efficiency and safety of the language core. If you take any of the language that was not developed in Rust and embedding it into a Rust application, you are losing a great deal of memory and other safety checks provided by the Rust.
- The language shall be a “sandboxed”, means the developer of the TSAK tool must be able to enable and disable language and library features that are not desirable for the TSAK application.
- The syntax of scripting language that are used in the TSAK tool must be familiar for the developers exposed to the procedural languages, but not necessary mimic any existing language.
- Standard library developed for the TSAK must have same efficiency and safety properties as other elements of the tool, provided by the Rust.

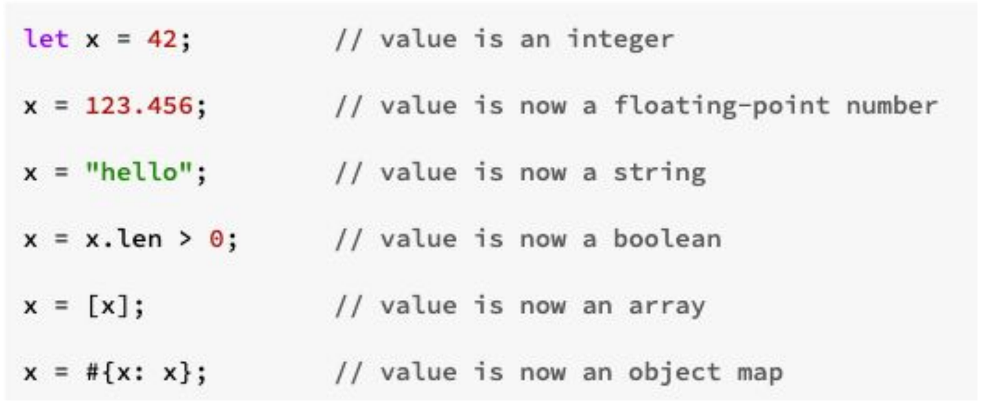
And here is just a few of the many features of RHAI scripting language core.

- RHAI scripting language core implements a very simple language with syntax similar to a JavaScript
- It does have a few external dependencies, none of them are breaking the safety checks provided by Rust
- RHAI have a minimal amount of “unsafe” code
- It is fairly efficient, providing 1M iterations in 0.14 seconds on single core 2.5Ghz Linux machine
- Scripts could be precompiled into an AST tree for even greater efficiency and repeated evaluation
- RHAI provides function and operator overloading mechanism
- RHAI provides closures functionality
- RHAI provides functionality for definition a custom operators and custom syntax
- RHAI language core could be “sandboxed” and can not change environment unless explicitly permitted
- It is guaranteed that it will not panic
- RHAI have embedded countermeasures against malicious attacks, such as stack overflow, runaway scripts and oversized data

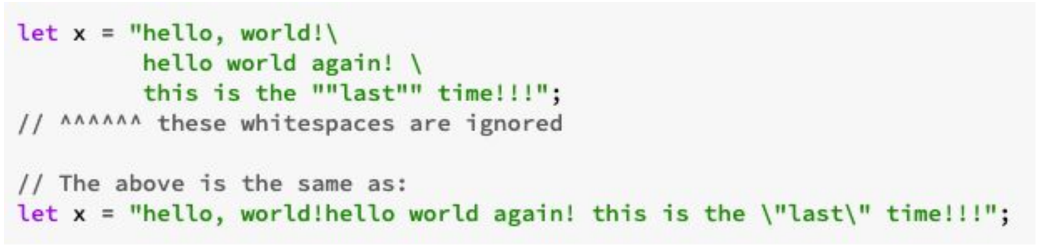
Now, let us review some of the functionalities of TSAK script, as they provided by RHAI scripting engine and TSAK standard library

TSAK script have a very flexible and dynamic data type systems.

- Integer
- Float
- String
- Timestamp
- Boolean
- Range
- Array
- Dictionary/Hash table
- Function
- Telemetry event and Event transaction
- Telemetry metric and Metric transaction
- Log entry
- Problem
- Perceptron/Neural network
- Grok pattern
- New Relic security findings
- NRQL query result



```
let x = 42;             // value is an integer

x = 123.456;            // value is now a floating-point number

x = "hello";            // value is now a string

x = x.len > 0;          // value is now a boolean

x = [x];                // value is now an array

x = #{x: x};            // value is now an object map
```



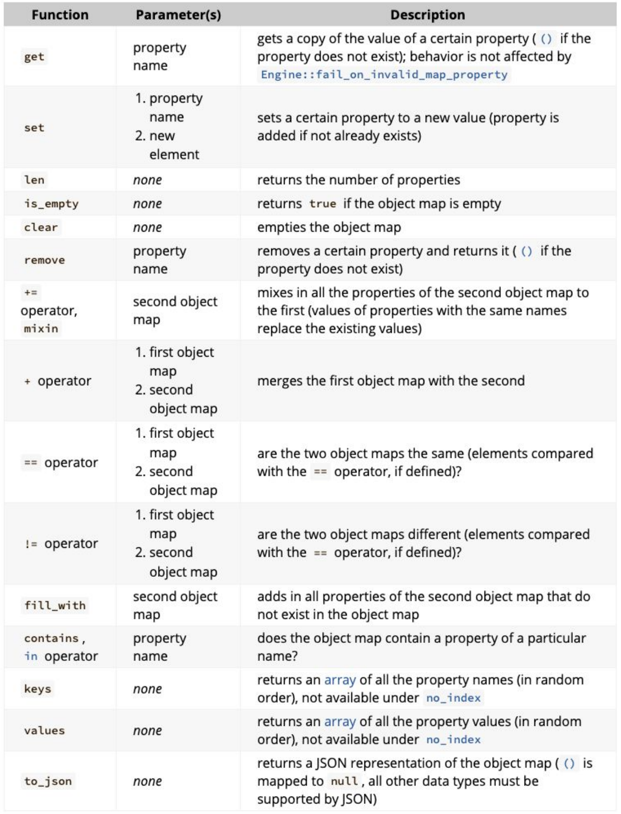
```
let x = "hello, world!\
        hello world again! \
        this is the ""last"" time!!!";
// ^^^^^^ these whitespaces are ignored

// The above is the same as:
let x = "hello, world!hello world again! this is the \"last\" time!!!";
```

TSAK script provides capabilities for working with arrays of data that you are expecting from a modern languages.

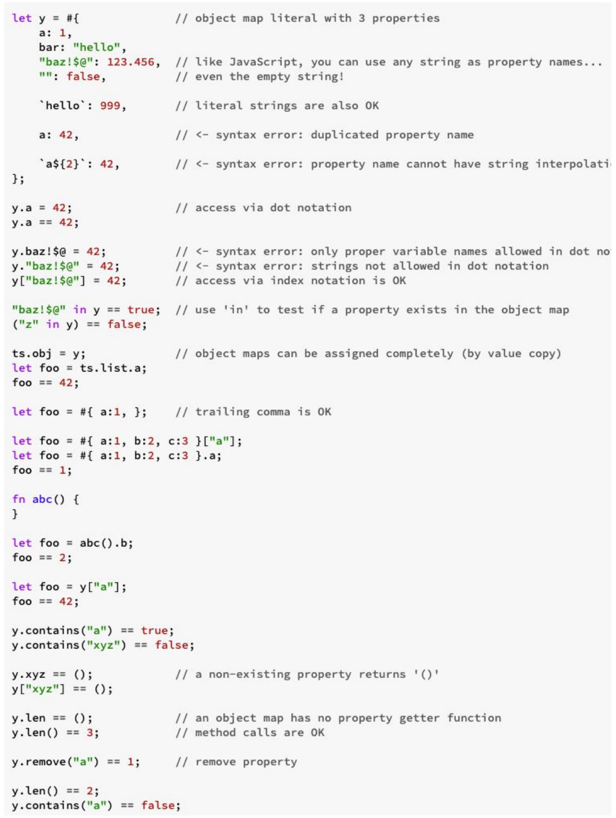
All array operators that you are expecting, including array definition, access to and manipulation with array elements, testing if specific data is present within array, interaction against array elements and many other.

```
let y = [2, 3];             // y == [2, 3]

let y = [2, 3,];            // y == [2, 3]

y.insert(0, 1);             // y == [1, 2, 3]

y.insert(999, 4);           // y == [1, 2, 3, 4]

y.len == 4;

y[0] == 1;
y[1] == 2;
y[2] == 3;
y[3] == 4;

(1 in y) == true;           // use 'in' to test if an element exists in the array

(42 in y) == false;         // 'in' uses the 'contains' function, which uses the
                            // '==' operator (that users can override)
                            // to check if the target element exists in the array

y.contains(1) == true;      // the above de-sugars to this

y[1] = 42;                  // y == [1, 42, 3, 4]

(42 in y) == true;

y.remove(2) == 3;           // y == [1, 42, 4]

y.len == 3;

y[2] == 4;                  // elements after the removed element are shifted

ts.list = y;                // arrays can be assigned completely (by value copy)

ts.list[1] == 42;

[1, 2, 3][0] == 1;          // indexing on array literal

[1, 2, 3][-1] == 3;         // negative position counts from the end
```

Hash tables or Dictionaries are the common data types in the modern languages and TSAK script provides full range of Hash table related capabilities and functions.

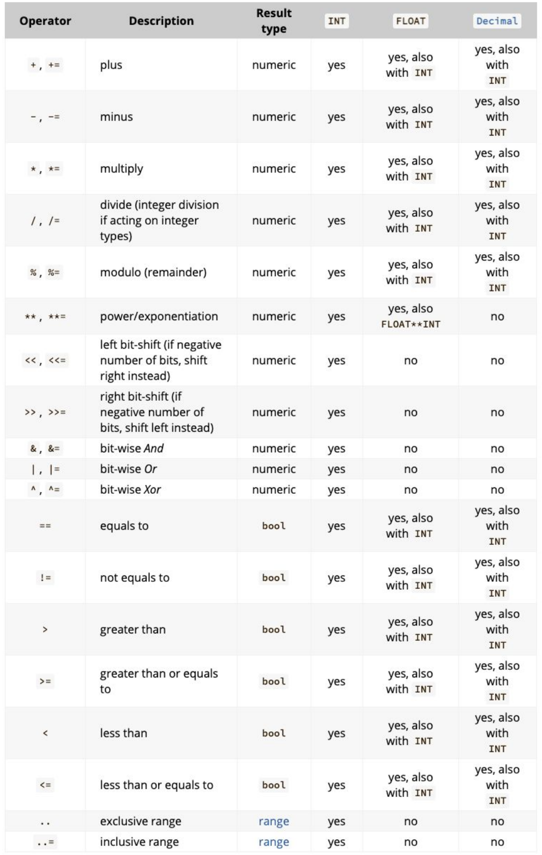


Function	Parameter(s)	Description
`get`	property name	gets a copy of the value of a certain property (`()` if the property does not exist); behavior is not affected by `Engine::fail_on_invalid_map_property`
`set`	1. property name 2. new element	sets a certain property to a new value (property is added if not already exists)
`len`	*none*	returns the number of properties
`is_empty`	*none*	returns `true` if the object map is empty
`clear`	*none*	empties the object map
`remove`	property name	removes a certain property and returns it (`()` if the property does not exist)
`+=` operator, `mixin`	second object map	mixes in all the properties of the second object map to the first (values of properties with the same names replace the existing values)
`+` operator	1. first object map 2. second object map	merges the first object map with the second
`==` operator	1. first object map 2. second object map	are the two object maps the same (elements compared with the `==` operator, if defined)?
`!=` operator	1. first object map 2. second object map	are the two object maps different (elements compared with the `==` operator, if defined)?
`fill_with`	second object map	adds in all properties of the second object map that do not exist in the object map
`contains`, `in` operator	property name	does the object map contain a property of a particular name?
`keys`	*none*	returns an array of all the property names (in random order), not available under `no_index`
`values`	*none*	returns an array of all the property values (in random order), not available under `no_index`
`to_json`	*none*	returns a JSON representation of the object map (`()` is mapped to `null`, all other data types must be supported by JSON)

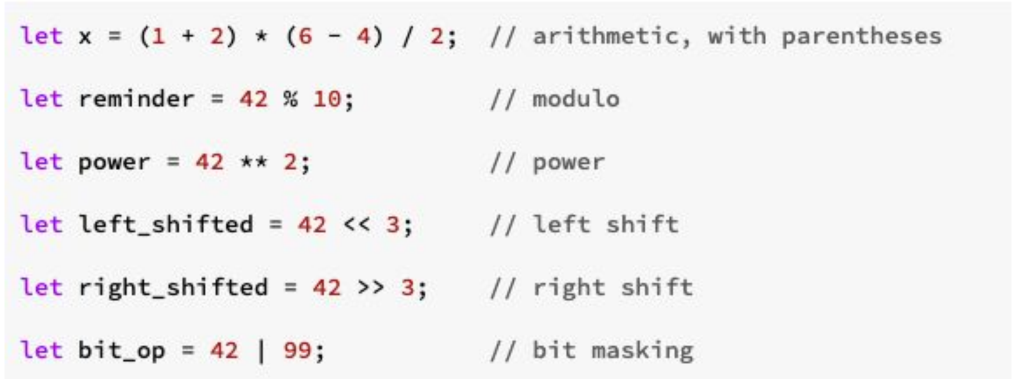


```
let y = #{              // object map literal with 3 properties
    a: 1,
    bar: "hello",
    "baz!$@": 123.456,  // like JavaScript, you can use any string as property names...
    "": false,          // even the empty string!

    `hello`: 999,       // literal strings are also OK

    a: 42,              // <- syntax error: duplicated property name

    `a${2}`: 42,        // <- syntax error: property name cannot have string interpolati
};

y.a = 42;               // access via dot notation
y.a == 42;

y.baz!$@ = 42;          // <- syntax error: only proper variable names allowed in dot no
y."baz!$@" = 42;        // <- syntax error: strings not allowed in dot notation
y["baz!$@"] = 42;       // access via index notation is OK

"baz!$@" in y == true;  // use 'in' to test if a property exists in the object map
("z" in y) == false;

ts.obj = y;             // object maps can be assigned completely (by value copy)
let foo = ts.list.a;
foo == 42;

let foo = #{ a:1, };    // trailing comma is OK

let foo = #{ a:1, b:2, c:3 }["a"];
let foo = #{ a:1, b:2, c:3 }.a;
foo == 1;

fn abc() {
}

let foo = abc().b;
foo == 2;

let foo = y["a"];
foo == 42;

y.contains("a") == true;
y.contains("xyz") == false;

y.xyz == ();            // a non-existing property returns '()'
y["xyz"] == ();

y.len == ();            // an object map has no property getter function
y.len() == 3;           // method calls are OK

y.remove("a") == 1;     // remove property

y.len() == 2;
y.contains("a") == false;
```

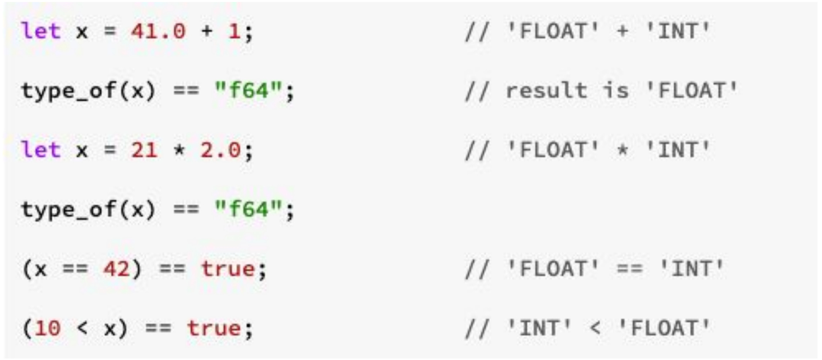
TSAK script provides comprehensive set of arithmetic operators.



Operator	Description	Result type	`INT`	`FLOAT`	`Decimal`
`+`, `+=`	plus	numeric	yes	yes, also with `INT`	yes, also with `INT`
`-`, `-=`	minus	numeric	yes	yes, also with `INT`	yes, also with `INT`
`*`, `*=`	multiply	numeric	yes	yes, also with `INT`	yes, also with `INT`
`/`, `/=`	divide (integer division if acting on integer types)	numeric	yes	yes, also with `INT`	yes, also with `INT`
`%`, `%=`	modulo (remainder)	numeric	yes	yes, also with `INT`	yes, also with `INT`
`**`, `**=`	power/exponentiation	numeric	yes	yes, also `FLOAT**INT`	no
`<<`, `<<=`	left bit-shift (if negative number of bits, shift right instead)	numeric	yes	no	no
`>>`, `>>=`	right bit-shift (if negative number of bits, shift left instead)	numeric	yes	no	no
`&`, `&=`	bit-wise *And*	numeric	yes	no	no
`|`, `|=`	bit-wise *Or*	numeric	yes	no	no
`^`, `^=`	bit-wise *Xor*	numeric	yes	no	no
`==`	equals to	`bool`	yes	yes, also with `INT`	yes, also with `INT`
`!=`	not equals to	`bool`	yes	yes, also with `INT`	yes, also with `INT`
`>`	greater than	`bool`	yes	yes, also with `INT`	yes, also with `INT`
`>=`	greater than or equals to	`bool`	yes	yes, also with `INT`	yes, also with `INT`
`<`	less than	`bool`	yes	yes, also with `INT`	yes, also with `INT`
`<=`	less than or equals to	`bool`	yes	yes, also with `INT`	yes, also with `INT`
`..`	exclusive range	range	yes	no	no
`..=`	inclusive range	range	yes	no	no



```
let x = (1 + 2) * (6 - 4) / 2;  // arithmetic, with parentheses

let reminder = 42 % 10;         // modulo

let power = 42 ** 2;            // power

let left_shifted = 42 << 3;     // left shift

let right_shifted = 42 >> 3;    // right shift

let bit_op = 42 | 99;           // bit masking
```



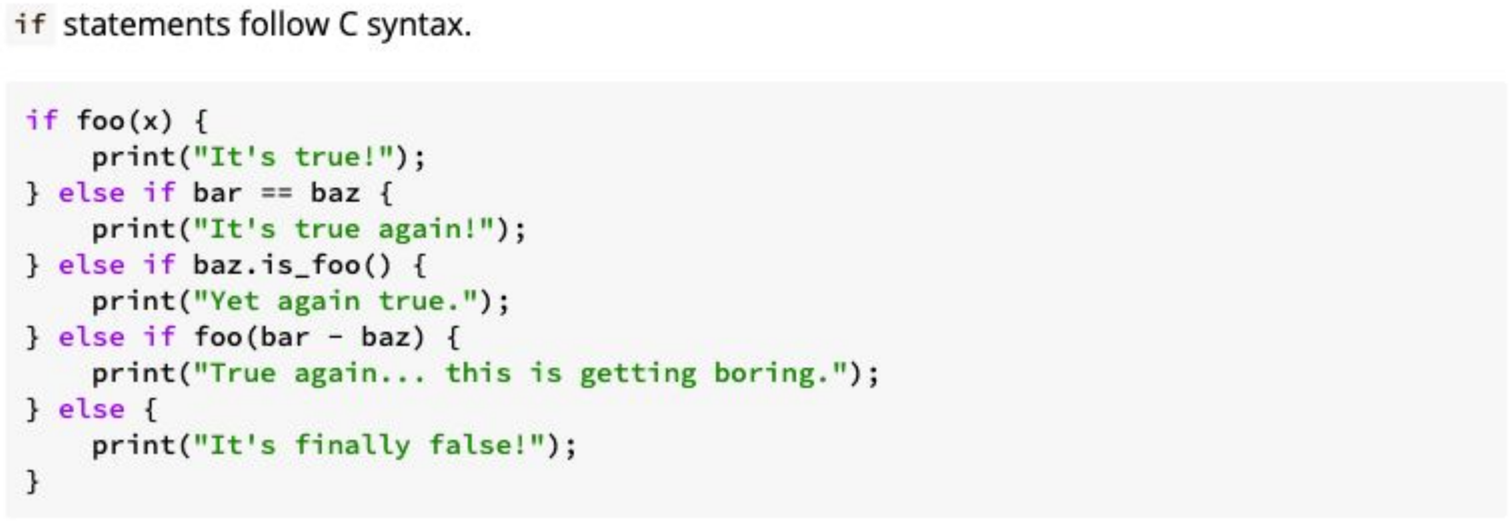
```
let x = 41.0 + 1;               // 'FLOAT' + 'INT'

type_of(x) == "f64";            // result is 'FLOAT'

let x = 21 * 2.0;               // 'FLOAT' * 'INT'

type_of(x) == "f64";

(x == 42) == true;              // 'FLOAT' == 'INT'

(10 < x) == true;               // 'INT' < 'FLOAT'
```

TSAK script provides also full set of logic operators.

Operator	Description (`x` *operator* `y`)	`x`, `y` same type or are numeric	`x`, `y` different types
`==`	`x` is equals to `y`	error if not defined	`false` if not defined
`!=`	`x` is not equals to `y`	error if not defined	`true` if not defined
`>`	`x` is greater than `y`	error if not defined	`false` if not defined
`>=`	`x` is greater than or equals to `y`	error if not defined	`false` if not defined
`<`	`x` is less than `y`	error if not defined	`false` if not defined
`<=`	`x` is less than or equals to `y`	error if not defined	`false` if not defined

```
42 == 42.0;          // true

42.0 == 42;          // true

42.0 > 42;           // false

42 >= 42.0;          // true

42.0 < 42;           // false
```

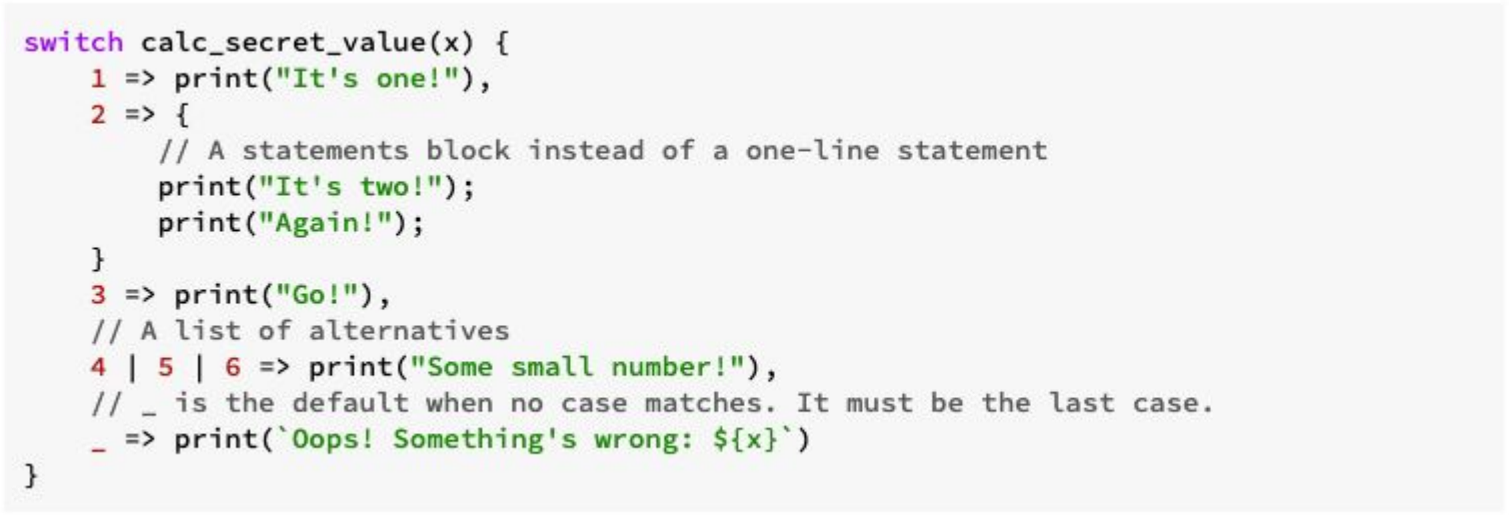
TSAK script do have an if operator. As expected.



`if` statements follow C syntax.

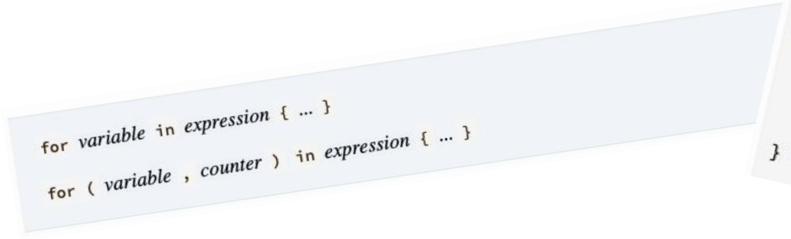
```
if foo(x) {
    print("It's true!");
} else if bar == baz {
    print("It's true again!");
} else if baz.is_foo() {
    print("Yet again true.");
} else if foo(bar - baz) {
    print("True again... this is getting boring.");
} else {
    print("It's finally false!");
}
```

TSAK script do have a switch operator. Mechanics of the switch operator is most close to a functional match operator.

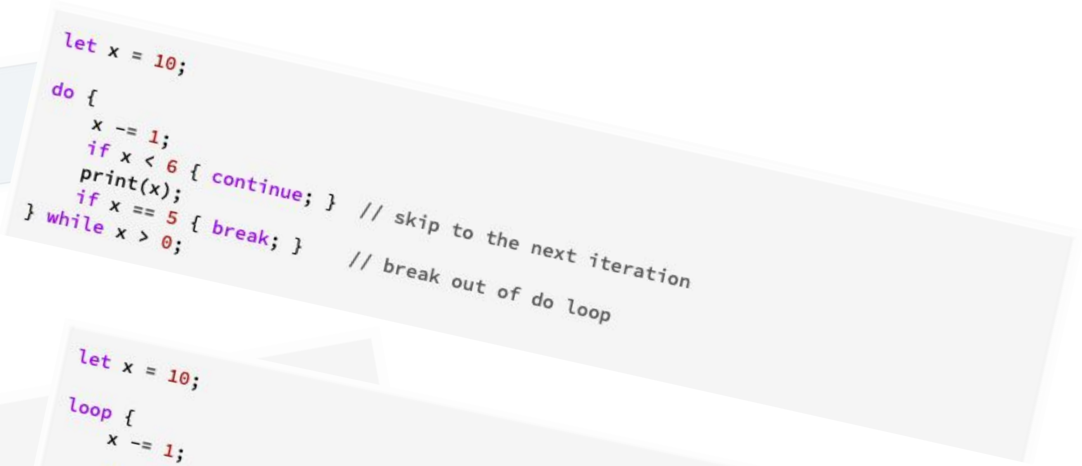


```
switch calc_secret_value(x) {
    1 => print("It's one!"),
    2 => {
        // A statements block instead of a one-line statement
        print("It's two!");
        print("Again!");
    }
    3 => print("Go!"),
    // A list of alternatives
    4 | 5 | 6 => print("Some small number!"),
    // _ is the default when no case matches. It must be the last case.
    _ => print(`Oops! Something's wrong: ${x}`)
}
```

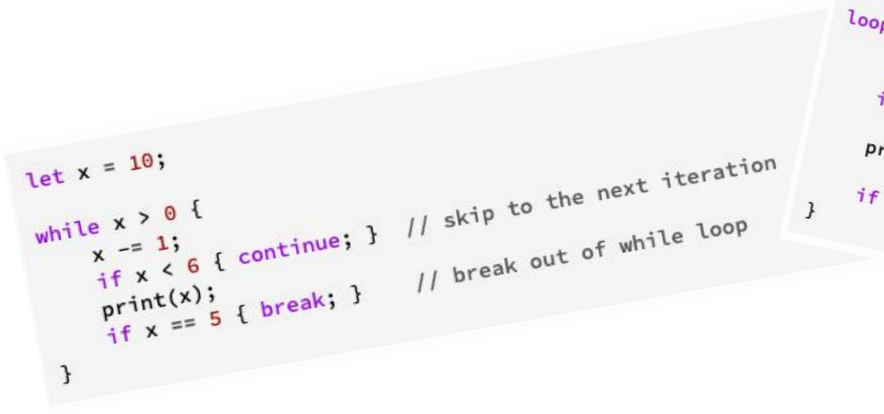
TSAK script a four types of the loop operators: for/do/while/loop.



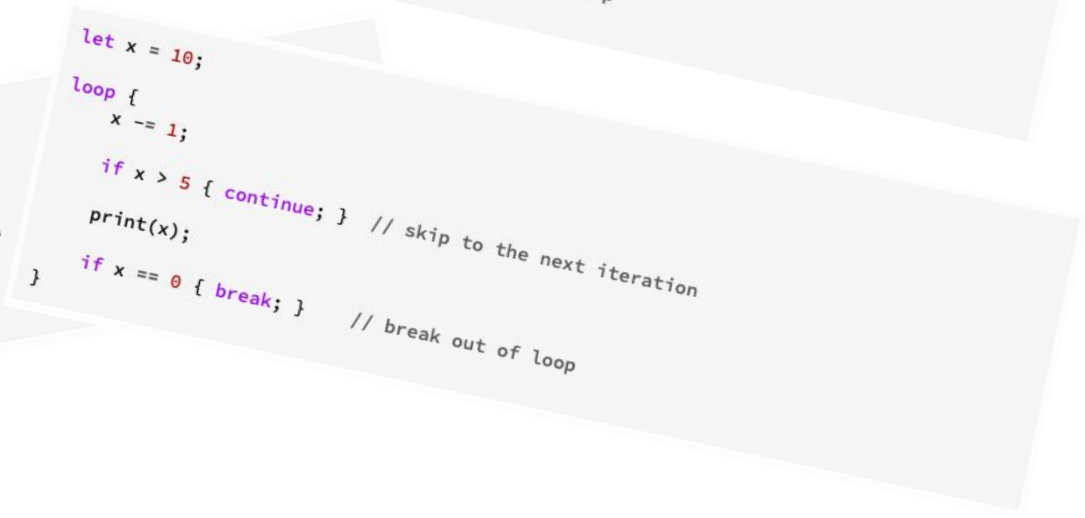
```
for variable in expression { ... }
for ( variable , counter ) in expression { ... }
```



```
let x = 10;
do {
    x -= 1;
    if x < 6 { continue; }  // skip to the next iteration
    print(x);
    if x == 5 { break; }    // break out of do loop
} while x > 0;
```



```
let x = 10;

while x > 0 {
    x -= 1;
    if x < 6 { continue; }  // skip to the next iteration
    print(x);
    if x == 5 { break; }    // break out of while loop
}
```



```
let x = 10;
loop {
    x -= 1;

    if x > 5 { continue; }  // skip to the next iteration

    print(x);

    if x == 0 { break; }    // break out of loop
}
```

Error handling is implemented through commonly used throw/catch mechanism.

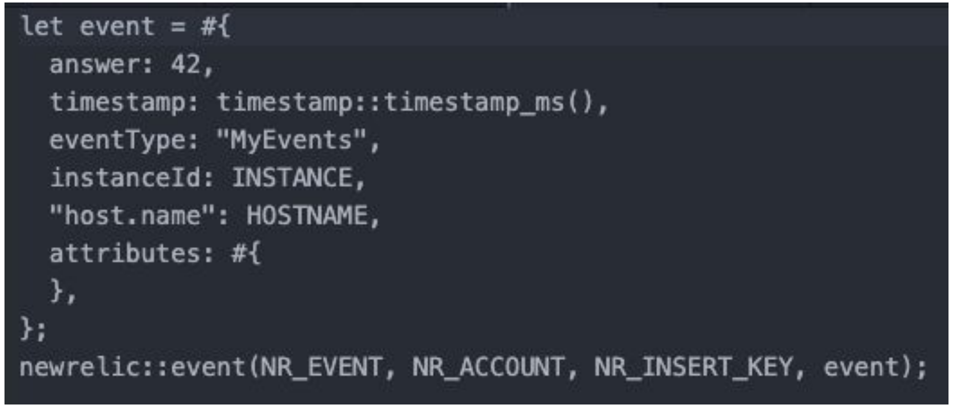
```
fn code_that_throws() {
    throw 42;
}

try
{
    code_that_throws();
}
catch (err)         // 'err' captures the thrown exception value
{
    print(err);     // prints 42
}
```

In TSAK script, functions are defined in a pretty familiar way.

```
fn add(x, y) {
    x + y
}

fn sub(x, y,) {     // trailing comma in parameters list is OK
    x - y
}

add(2, 3) == 5;

sub(2, 3,) == -1;   // trailing comma in arguments list is OK
```

How to solve common problems using TSAK tool.

How to send a single event using Hash table event representation.

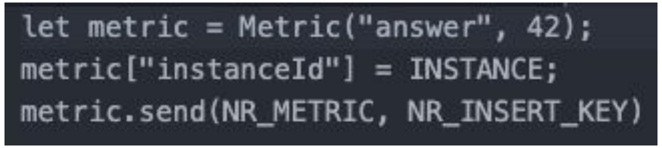


```
let event = #{
  answer: 42,
  timestamp: timestamp::timestamp_ms(),
  eventType: "MyEvents",
  instanceId: INSTANCE,
  "host.name": HOSTNAME,
  attributes: #{
  },
};
newrelic::event(NR_EVENT, NR_ACCOUNT, NR_INSERT_KEY, event);
```

Internally, all Events, Metrics, Logs are represented by a Hash table. So, you can create a Hash table, where event will be having a very familiar, close to a raw event API form. Then you can use function “event” from standard library module “newrelic” to send a single event. Information about account, event type and New Relic key are passed by default ether through environment variables or through CLI parameters, but you can obtain them in any other way.

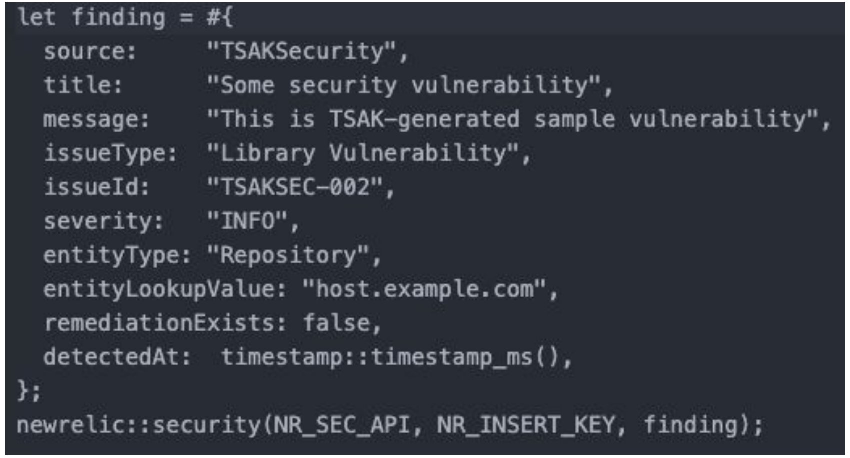
How to send a single metric using Metric object.

Metrics, Events and Logs could be formatted and sent through creation of the appropriate objects. This is a more or less a “syntax sugar” over creation of the telemetry metric data through Hash table. But it does provide a convenient interface for setting a metric properties and method “.send” of the type Metric will send a single metric to a New Relic Metrics API frontend.



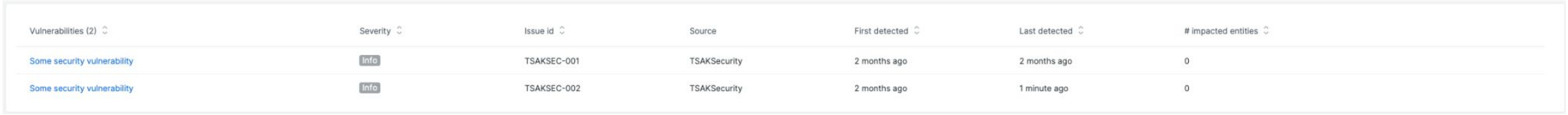
```
let metric = Metric("answer", 42);
metric["instanceId"] = INSTANCE;
metric.send(NR_METRIC, NR_INSERT_KEY)
```

How to send a security finding through Security API.



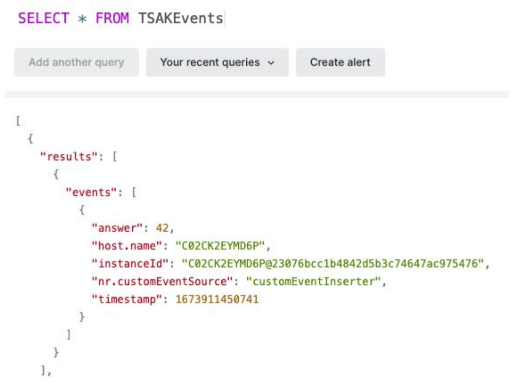
```
let finding = #{
  source:     "TSAKSecurity",
  title:      "Some security vulnerability",
  message:    "This is TSAK-generated sample vulnerability",
  issueType:  "Library Vulnerability",
  issueId:    "TSAKSEC-002",
  severity:   "INFO",
  entityType: "Repository",
  entityLookupValue: "host.example.com",
  remediationExists: false,
  detectedAt:  timestamp::timestamp_ms(),
};
newrelic::security(NR_SEC_API, NR_INSERT_KEY, finding);
```

You can use TSAK tool to create an integration with external security tools and send discovered vulnerabilities through New Relic Security API. As with events, logs and metrics, Hash tables are the core data type for the standard library API and you can conveniently use function “security” from the module “newrelic” of standard TSAK library to send finding to Security API.



Vulnerabilities (2)	Severity	Issue id	Source	First detected	Last detected	# impacted entities
Some security vulnerability	Info	TSAKSEC-001	TSAKSecurity	2 months ago	2 months ago	0
Some security vulnerability	Info	TSAKSEC-002	TSAKSecurity	2 months ago	1 minute ago	0

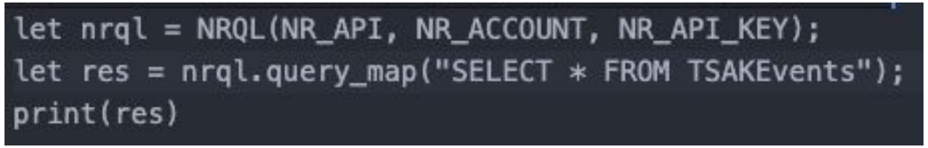
How to send an NRQL query and get the result.



```
SELECT * FROM TSAKEvents
```

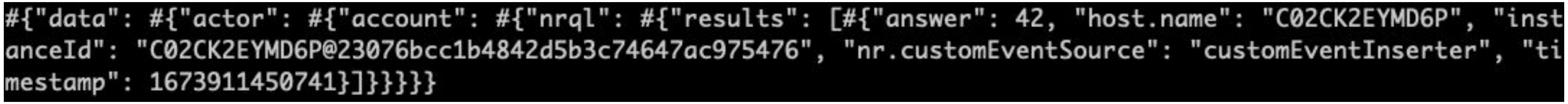
```
[
  {
    "results": [
      {
        "events": [
          {
            "answer": 42,
            "host.name": "C02CK2EYMD6P",
            "instanceId": "C02CK2EYMD6P@23076bcc1b4842d5b3c74647ac975476",
            "nr.customEventSource": "customEventInserter",
            "timestamp": 1673911450741
          }
        ]
      }
    ],
```

Imagine, that you do have an events of the type TSAKEvents that you do like to query.



```
let nrql = NRQL(NR_API, NR_ACCOUNT, NR_API_KEY);
let res = nrql.query_map("SELECT * FROM TSAKEvents");
print(res)
```

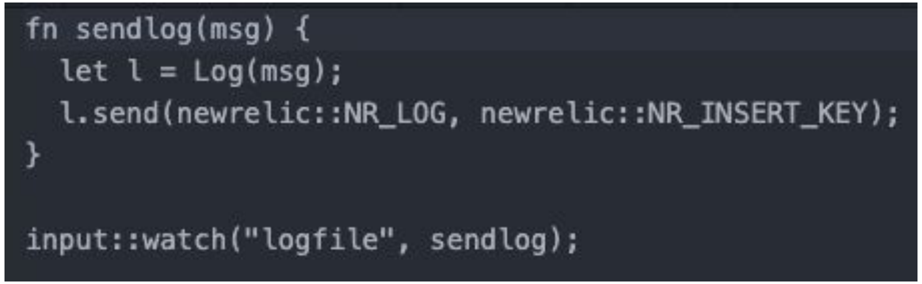
TSAK scripts provides you with convenient ways to query NRDB and bring results to the script for further processing, analysis and perhaps generating more data.



```
#{"data": #{"actor": #{"account": #{"nrql": #{"results": [#{"answer": 42, "host.name": "C02CK2EYMD6P", "inst
anceId": "C02CK2EYMD6P@23076bcc1b4842d5b3c74647ac975476", "nr.customEventSource": "customEventInserter", "ti
mestamp": 1673911450741}]}}}}}
```

This call sends NRQL query and return data in Hash Table format. Other options will be to load data into a DataFrame that is also supported by TSAK script.

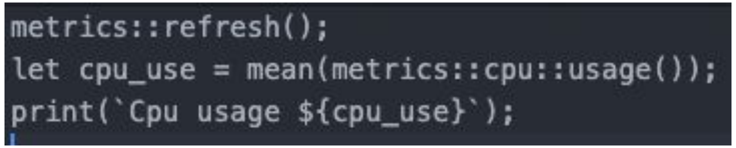
How to use TSAK tool as a log file processor.



```
fn sendlog(msg) {
  let l = Log(msg);
  l.send(newrelic::NR_LOG, newrelic::NR_INSERT_KEY);
}

input::watch("logfile", sendlog);
```

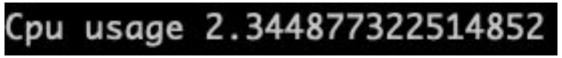
So far, I’ve show to you how you can feed New Relic with data and query New Relic for some data that you submit before. But now, let me show you how you can use TSAK tool as an instrument that can replace 3-rd party log processing instruments and watch and send log entries that stored in the local log file. File rotation detection is supported, of course. In this example, you are using function “watch” from the module “input” of TSAK standard library that is calling user-defined function “sendlog” for every new log entry. In this function, we are using Log data type to create a New Relic log entry and send it to a Log API frontend using “.send” method.

Can we use TSAK tool as an Infrastructure agent ? Absolutely !



```
metrics::refresh();
let cpu_use = mean(metrics::cpu::usage());
print(`Cpu usage ${cpu_use}`);
```

TSAK standard library provided an access to the localhost telemetry, including cpu, disk, network, memory utilization as well as other parameters commonly collected by New Relic, or OTel or Prometheus agents. First, you have to refresh local metrics using function “refresh” from module “metrics” of TSAK standard library. Next, we obtain a sample of cpu usage telemetry data and computing a mean value from the array of samples.

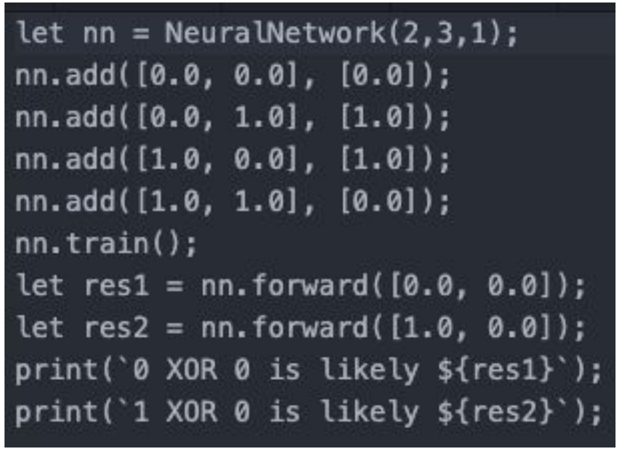


```
Cpu usage 2.344877322514852
```

How to train and use Neural Network capabilities that is a part of the TSAK standard library.

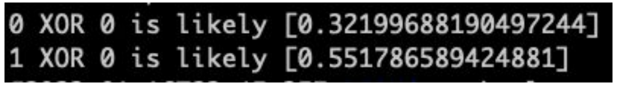
Neural Networks are proven to be a valuable tool for data analysis and patent finding and matching. TSAK standard library includes a simple Perceptron that will help with most common pattern matching tasks. And as “hello world” of ML, we will be using pretty common example of computing XOR algorithm that is using ML.

First, we will create a variable of type “NeuralNetwork”. This type is supported by the TSAK standard library. Our Perceptron will have a two inputs, one output and three internal.



```
let nn = NeuralNetwork(2,3,1);
nn.add([0.0, 0.0], [0.0]);
nn.add([0.0, 1.0], [1.0]);
nn.add([1.0, 0.0], [1.0]);
nn.add([1.0, 1.0], [0.0]);
nn.train();
let res1 = nn.forward([0.0, 0.0]);
let res2 = nn.forward([1.0, 0.0]);
print(`0 XOR 0 is likely ${res1}`);
print(`1 XOR 0 is likely ${res2}`);
```

Next step, will be adding training data to created Perceptron and train it. The quality of Neural Network will depend heavily on a size and quality of a training data. Since we have only four data set in our training data, our outcome will not be great, but we will get an identifiable results.



```
0 XOR 0 is likely [0.32199688190497244]
1 XOR 0 is likely [0.551786589424881]
```

CONCLUSIONS.

In this presentation, I’ve hopefully bring enough information that is defining the place of the TSAK tool in the enterprise and a customer environments.