LANGUAGE REFERENCE MANUAL: SIFT


```
Anne Zepecki(Manager)
Lama Abdullah Rashed(Tester)
Jose Antonio Ramos(Language Guru)
Rishav Kumar (System Architect)
Suryanarayana Venkata Akella (System Architect)
```


1. Introduction

Sift is a programming language designed to optimize text processing and filtering for natural language processing (NLP) applications. Sift draws inspiration from the elegance of C's syntax while adding its own unique flair. This imperative language is designed for users with some programming experience as knowledge of C or a similar language will allow users to instantly recognize and understand the basic structure of Sift. However, it is the text-specific processing tools that make Sift stand out as the best language to use for NLP purposes; piping, filtering, and basic tokenization functionality give users the tools they need to seamlessly integrate Sift into their existing or future NLP workflows.

2. Lexical Conventions

2.1. Tokens

Sift includes five types of tokens: keywords, identifiers, literals, operators, and separators. Separators (blanks, tabs, newlines, and comments) are typically ignored except to separate tokens.

Separators are defined as follows:

```
' ' /* This is a single space */
'\t' /* This is a horizontal tab */
'\r' /* This is a carriage return */
'\n' /* This is a newline */
```

Comments begin with `/*` and close with `*/`; they cannot be nested and may not appear within literals. Like whitespaces, comments are ignored.

```
/* This is an example of a comment */
```

2.2. Identifiers

Identifiers are composed of a sequence of letters, digits, and the _ punctuation. The first character of any identifier must be in the range [a-z A-Z] (cannot be a digit). Upper and lowercase letters are distinct. Identifiers may not have the same value as a keyword.

```
/* Below see the regex pattern for valid identifiers */
['a' - 'z'] | ['A' - 'Z'] | ['0' - '9'] | '_'
```

2.3. Keywords

Keywords are identifiers with specific use and meaning within Sift and may not be used except for their intended defined use.

See list of keywords:

```
bool
char
int
float
str
sym
if
else
from
import
def
list
return
map
pure
set
arr
dict
for
while
switch
case
break
continue
lambda
default
tup
```

2.4. Literals

There are several types of literals that may be defined in Sift.

int Literal

An int literal is a sequence of digits from `['0'-'9']+`.

float Literal

A float literal includes an int component, a decimal point (`'.'`) a fractional component, and an optional `'E'` with an int exponent. All int and fractional components are digits in `['0'-'9']`.

Thus a float is defined as `['0'-'9']+['.']['0'-'9']+`

bool Literal

A bool literal can hold two possible sequences of characters -- `true` or `false`.

char Literal

A char literal is a single character surrounded by single quotation marks. Special chars are represented with a backslash escape sequence preceding the character.

A char can be any valid ASCII character.

str Literal

A str literal is a sequence of chars surrounded by double quotes. The valid set includes anything that can be represented as a char literal.

sym literal

A sym literal is a sequence of chars surrounded by double quotes. The valid set includes anything that can be represented as a char literal. sym literals are final and their values cannot be modified.

2.5. Operators

Operators are elements reserved for use by Sift and cannot be used for any purposes other than their reserved purpose. Please refer to the Operators and the Expressions sections for more details.

Assignment: `=`

Equivalence: `==, !=, <, >, <=, =>`

Arithmetic: `+, -, *, /, %`

Logical: `&&, ||, !`

Special Functionality: `|>`

2.6. Separators

Separators are elements reserved for use by Sift and cannot be used for any purposes other than their reserved purpose. They indicate separation between tokens. As mentioned above, whitespace is also considered a valid separator.

```
.
,
:
;
(
)
[
]
{
}
```

3. Types

3.1. Primitive Data Types

bool

bool stores either true or false values

```
bool x = true;
```

char

char stores one character in 8 bits. Signified by single quotation marks.

```
char x = 'a';
```

int

int type stores whole number value in 32 bits

```
int x = 4;
```

float

float type stores fractional number value in 32 bits

```
float x = 1.0;
```

str

str type stores a sequence of chars in UTF-8 format. Signified by double quotation marks. str types are **mutable**.

```
str x = "sift";
```

sym

sym type stores a sequence of chars in UTF-8 format. Signified by double quotation marks. sym types are **immutable**.

```
sym x = "sift";
```

3.2. Non-Primitive Data Types

arr

arr type stores a fixed-size array that contains a decalared primitive or non-primitive data type as its elements.

```
arr<int> test = [1, 2, 3];
```

list

list type stores a doubly-linked list that can change size dynamically. list elements are a declared primitive or non-primitve data type.

```
list<sym> test = ["test1", "test2", "test3"];
```

tup

tup type stores elements that may be from different data types. tup may contain no elements ("empty") or many elements.

```
tup test = (0, 'a', "test");
```

set

set type stores an unordered collection of declared primitive or non-primitive elements. A set may not contain duplicate elements.

```
set<int> = (1, 2);
```

dict

dict type stores a mapping of one declared primitive or non-primitive data-type to any other declared primitive or non-primitive data types. The `key` values of a dict are a set (and therefore may not contain duplicates).

```
dict<str, str> = {"k1": "v1", "k2": "v2"}
```

3.3. Type Qualifiers

pure

The `pure` keyword is used to denote that a function is pure (defined as a function that returns the same output type(s) as input type(s) and does not cause any side effects).

```
def pure f1(int x):
```

lambda

The `lambda` keyword is used to denote an anonymous function.

```
lambda x : x + 1;
```

3.4. Type Cast

For safety reasons, Sift does not do any implicit type cast. Explicit type cast is supported using `(type)` in front of the variable that should be cast.

```
str a = "test";
sym b = (sym) a;
```

4. Operators

4.1. Multiplicative Operators

The multiplicative operators are `*, /, %` and group from left to right.

4.1.1. expression * expression

The binary `*` operator indicates multiplication. If both operands are int or char, the result is int; if one is int or char and one float, the former is converted to float, and the result is float; if both are float, the result is float. No other combinations are allowed.

```
int x = 10;
int y = 8;
int z = x * y;
```

4.1.2. expression / expression

The binary `/` operator indicates division. The same type considerations as for multiplication apply.

```
int x = 10;
int y = 8;
int z = x / y;
```

4.1.3. expression % expression

The binary `%` operator yields the remainder from the division of the first expression by the second. Both operands must be int or char, and the result is int. In the current implementation, the remainder has the same sign as the dividend.

```
int x = 10;
int y = 8;
int z = x % y;
```

4.2. Additive Operators

The additive operators `+` and `–` group left-to-right.

4.2.1. expression + expression

The result is the sum of the expressions. If both operands are int or char, the result is int. If both are float, the result is float. If one is char or int and one is float, the former is converted to float and the result is float. No other type combinations are allowed.

```
int x = 10;
int y = 8;
int z = x + y;
```

4.2.2. expression – expression

The result is the difference of the operands. If both operands are int, char, or float, the same type considerations as for + apply.

```
int x = 10;
int y = 8;
int z = x + y;
```

4.3. Assignment Operators

4.3.1. lvalue = expression

The operator = assigns an expression to a variable. It assigns value on right of operator to the left. The value of the expression replaces that of the object referred to by the lvalue. The operands need not have the same type, but both must be int, char, float.

```
int x = 5;
```

4.3.2. lvalue =+ expression

```
int x = 5;
int y =+ x;
```

4.3.3. lvalue =– expression

```
int x = 5;
int y =- x;
```

4.3.4. lvalue =* expression

```
int x = 5;
int y =* x;
```

4.3.5. lvalue =/ expression

```
int x = 5;
int y =/ x;
```

4.3.6. lvalue =% expression

```
int x = 5;
int y =% x;
```

The behavior of an expression of the form `E1 =op E2` may be inferred by taking it as equivalent to `E1 = E1 op E2`; however, E1 is evaluated only once.

4.4. Relational Operators

The relational operators are <, <=, >, >= . They all have the same precedence. They group from left to right.

4.4.1. expression < expression

```
bool less = 5 < 6;
```

4.4.2. expression > expression

```
bool greater = 5 > 6;
```

4.4.3. expression <= expression

```
bool less_than_equal = 5 < 6;
```

4.4.4. expression >= expression

```
bool greater_than_equal = 5 < 6;
```

The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. Operand conversion is exactly the same as for the + operator.

4.5. Pipe Operator |>

There is also a pipe operator |> much like pipe in ocaml which applies a function to another.

Example Syntax

```
str x = "hellopeople";
str s = f |> g |> x;
```

4.7. Equality Operators

4.7.1. expression == expression

```
str hello = "hello";
str world = "world";
bool i_am_false = hello == world;
string x = "hellopeople"
string s = f |> g |> x
```

4.7.2. expression != expression

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus a<b == c<d is 1 whenever a<b and c<d have the same truth-value).

4.8. Example Syntax

```
str hello = "hello";
str world = "world";
bool i_am_true = hello != world;
```

4.9. Logical Operators

The logical operators are &&, ||, !.

4.9.1. expression && expression

The && operator returns 1 if both its operands are non-zero, 0 otherwise. && guarantees left-to-right evaluation; moreover the second operand is not evaluated if the first operand is 0. The operands need not have the same type, but each must have one of the fundamental types.

```
int x = 5;
int y = 5;
int z = 5;

bool i_am_true = (x == y && x == z);
```

4.9.2. expression || expression

The || operator returns 1 if either of its operands is non-zero, and 0 otherwise. , || guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the value of the first operand is non-zero. The operands need not have the same type, but each must have one of the fundamental types.

```
int x = 5;
int y = 5;
int z = 10;

bool i_am_true = (x == y || x == z);
```

4.9.3. !expression

The ! operator returns 1 if operand is zero, and 0 otherwise.

```
bool i_am_true = true;
bool i_am_false = !i_am_true;
```

5. Statements & Expressions

5.1. Variable Declaration

In sift, variables are declared by stating their data types followed by variable name and initializer expression. The scope of a variable is its lifetime in the program.

Syntax:

```
int x = 5;

str client = “David”;
```

5.2. Literal expressions

A literal expression is a literal (i.e.: int,str,sym) followed by a semicolon.

Syntax:

```
"hello,world!";
2;
```

```
2.1;
'a';
true;
```

5.3. List xpressions

A list expression can contain zero ("empty") or more values; the values are placed in a comma-separated list inside brackets. The values inside the list should have the same data type.

Syntax:

```
list<str> x = [];
list<str> x = ["hello", " world "];
list<int> x = [1,2,3];
```

5.4. Array expressions

An array expression can contain zero ("empty") or more values; the values are placed in a comma-separated list inside brackets. The values inside the list should have the same data type with a fixed size.

Syntax:

```
arr<str> x = [];
arr<str> x = ["hello", " world "];
arr<int> x = [1,2,3];
```

5.5. Dictionary expressions

A dictionary expression contains a series of keys and values enclosed in curly brackets. A colon is used to separate the key and value, and a comma is used to separate key and value pairs. Keys cannot be duplicated; the duplicated keys will be overwritten.

Syntax:

```
dict<str, str> x =
{
"1":“hello”,
"2":“world”
};
```

5.6. Set expressions

A set expression is written by enclosing zero or more elements in parentheses; with a comma that separates the elements. The elements inside the list should have the same data type, in no particular order. Sets cannot contain duplicate elements.

Syntax:

```
set<str> x = ("hello", " world ");
set<str> x = ( );
```

5.7. Tuple expressions

A tuple expression contains elements from different data types. The tuple can contain zero elements ("empty") or many elements.

Syntax:

```
tup test = (1,'a',"abc")
```

5.8. Control flow

5.8.1. Conditions and If Statements:

If/else

The `if` statement takes a conditional expression and executes the consequent block - enclosed in curly brackets - if the expression evaluates to true. The `if` statement can have an optional `else` section that executes its consequent block if the conditional expression evaluates to false. The `else if` can add another condition to evaluate; if the conditional expression is true, the consequent block will get executed. And the optional `else` statement will be skipped. If the `else if` conditional expression evaluates to false, then the optional else section get executed. The statement in each block must evaluate to the same unit type.

Syntax:

```
if (condition 1)
{
 // block of code to be executed if condition1 is true
}
else if (condition2)
{
  // block of code to be executed if condition1 is false and condition2 is
True
}
else
{
  // block of code to be executed if the condition1 is false and
condition2 is false
}
```

Switch

To specify many alternatives, we use `switch`. `switch` takes an expression inside parentheses, and compares the value of the expression with the values of each case, if there is a match, the associated block of code is executed. If there is no case match, the switch runs the `default` section ("if any"). In `switch`, each `case` keyword is followed by a label/value to be matched to, and a colon followed by a code block to be excuted if the case label matches the expression value.

Syntax:

```
switch (expression){
case constant1: // statements;
      break;
case constant2:// statements;
      break;
default:
      // default statements;
}
```

5.8.2. For loops

In Sift `for` loops are used to loop through arrays. A loop takes three statements separated by a semicolon. The first statement is the initialization statement, the second statement is a test expression, and the third statement is an update statement. In each iteration, the test expression is tested; if it's evaluated to true, the consequent block get executed, and the update expression is updated. However, if it's evaluated to false, the `for` loop is terminated.

Syntax:

```
for (statement1; statement2; statement3)
{
    // statements inside the body of loop
}
```

5.8.3. While

`while` loops evaluate the conditional value inside the parentheses; if it's evaluated to true, the consequent block gets executed. The conditional value is tested again: if it's evaluated to false the `while` loop is terminated; otherwise, the consequent block gets executed again.

```
while (condition)
{
  // code block to be executed
}
```

5.8.4. Break & Continue

`break` and `continue` expressions are used in loops to alter control flow. `break` is used to terminate the execution of a block of code. And the `continue` expression terminates the current iteration and cause the next iteration of the loop to run.

5.8.5. Function Calls

In Sift, functions are declared with the keyword `def`. The function should have a name and a consequent block of code to be executed when the function is called. A function can take zero, one or more arguments and return an expression of a certain type; the retuned type should be declared in the function header.

Syntax:

```
def list<(sym,int)> get_frequencies(list<sym> tokens) {
return ... ;
}
```

```
def hello() {
print(“Hello World!”);
}
```

6. Memory Management

As a text processing language, memory management is critical in Sift. This is why, unlike a language like C, Sift includes automatic memory management as a language feature. With our automatic memory management, it's faster to develop programs without being bothered about the low-level memory details. It also prevents the program from memory leaks and dangling pointers.

Sift uses generational garbage collection for automatic memory management and supports three generation. An object moves into an older generation whenever it survives a garbage collection process on its current generation.

The garbage collector keeps track of all objects in the memory. When a new object is created, it starts its life in the first generation. There is a predefined value for the theshold number of objects for each generation. If it exceeds that threshold, the garbage collector triggers a collection process. For objects that survive this process, they are moved into the older generation.

6.1. Using The Garbage Collector Module

Garbage collection is available to Sift users in the form of an importable module gc with a specific set of functionality. Programmers can change this default behavior of Garbage collection depending upon their available resources and requirement of their program.

You can check the configured thresholds of your garbage collector with the get_threshold() method:

```
>>> import gc;
>>> gc.get_threshold();
(1000, 300, 300)
```

By default, Sift has a threshold of 1000 objects for the first generation and 300 each for the second and third generations.

Check the number of objects in each generation with get_count() method:

```
>>> import gc;
>>> gc.get_count();
(572, 222, 109)
```

In the above example, we have 572 objects of first generation, 222 objects of second generation and 109 objects of third generation.

Trigger the garbage collection process manually

```
>>> import gc;
>>> gc.get_count();
(572, 222, 109)
>>> gc.collect();
```

User can set thresholds for triggering garbage collection by using the set_threshold() method in the gc module.

```
>>> import gc;
>>> gc.get_threshold();
(1000, 300, 300)
>>> gc.set_threshold(2000, 30, 30);
>>> gc.get_threshold();
(2000, 30, 30)
```

Increasing the threshold will reduce the frequency at which the garbage collector runs. This will improve the performance of your program but at the expense of keeping dead objects around longer.

7. Regular Expressions

For general text processing, Sift provides a module for working with regular expressions. When combined with filtering, pattern matching is a powerful tool for working with text data. To utilize regular expressions, users must import the `regex` module.

```
>>> import regex;
```

Regular expressions must be contained within a `str`, then passed to the methods contained in the `regex` module.

7.1. Regular Expression Syntax

We utilize the word "string" in this subsection to refer to a sequence of characters. We follow the regex syntax definition layed out by Russ Cox in *Regular Expression Matching Can Be Simple And Fast*:

https://swtch.com/~rsc/regexp/regexp1.html

Whenever a string *v* is in the language defined by some regular expression *e*, we say that *e matches v*.

The syntax for regular expressions is defined as such:

- The simplest regular expression is a single character. A single-character regular expression matches itself. There are meta-characters which do not match themselves: `*+?()|`. To match a metacharacter, escape it with the backslash character `\`.
- Two regular expressions may be *alternated* or *concatenated* to form a new regular expression: if e_1 matches *s* and e_2 matches *t*, then $e_1|e_2$ matches *s* or *t*, and e_1e_2 matches *st*.
- The metacharacters `*`, `+`, and `?` are repetition operators: *e** matches a sequence of zero or more strings, each of which match *e*; *e+* matches one or more; *e?* matches zero or one.

The operator precedence, from weakest to strongest binding:

1. alternation
2. concatenation
3. repetition operators

Explicit parentheses can be used to force different meanings,

7.2. Regular Expression Module

There are a few methods provided to match regular expressions with strings:

7.2.1. 1. match

The `match` method gets all the substrings in the input `str` or `sym` that match a given regular expression.a given regular expression. They are returned in a `list`:

There is a `str` variant and a `sym` variant:

- `list\<str\> match(str regular_expression, str text)`

```
>>> import regex;
>>> str expr = "(w|m)i*ld"
>>> str text = "The wiiiiiiiiiild wild cat lived in a mild climate.";
>>> regex.match(expr, text);
["wiiiiiiiiiild", "wild", "mild"]
```

- `list\<sym\> match(str regular_expression, sym text)`

```
>>> import regex;
>>> str expr = "(w|m)i*ld"
>>> sym text = "The wiiiiiiiiiild wild cat lived in a mild climate.";
>>> regex.match(expr, text);
["wiiiiiiiiiild", "wild", "mild"]
```

7.2.2. 2. test

The `test` method checks if a substring exists in the input `str` or `sym` that matches a given regular expression. Returns a `bool`.

There is a `str` variant and a `sym` variant:

- `bool test(str regular_expression, str text)`

```
>>> import regex;
>>> str expr = "h(i|ello)"
>>> str hello_text = "hello";
>>> regex.test(expr, hello_text);
true
>>> str ello_text = "ello";
>>> regex.test(expr, ello_text);
false
>>> str hi_text = "hihihihihihihi";
>>> regex.test(expr, hi_text);
true
```

- `bool test(str regular_expression, sym text)`

```
>>> import regex;
>>> str expr = "h(i|ello)"
>>> sym hello_text = "hello";
>>> regex.test(expr, hello_text);
true
>>> sym ello_text = "ello";
>>> regex.test(expr, ello_text);
false
>>> sym hi_text = "hihihihihihihi";
>>> regex.test(expr, hi_text);
true
```

7.2.3. 3. match_indices

The `match_indices` method gets all the leading indeces of the substrings in the input `str` or `sym` that match a given regular expression. They are returned in a `list`:

There is a `str` variant and a `sym` variant:

- `list\<int\> match_indices(str regular_expression, str text)`

```
>>> import regex;
>>> str expr = "(w|m)i*ld"
>>> str text = "The wiiiiiiiiiild wild cat lived in a mild climate.";
```

```
>>> regex.match_indices(expr, text);
[4, 18, 38]
```

- `list\<int\> match_indices(str regular_expression, sym text)`

```
>>> import regex;
>>> str expr = "(w|m)i*ld"
>>> sym text = "The wiiiiiiiiiild wild cat lived in a mild climate.";
>>> regex.match_indices(expr, text);
[4, 18, 38]
```

8. NLP Features

As a text processing language, Sift provides some Natural Language Processing (NLP) functionality to aid in processing natural language text data. To access nlp functionality, users must import the `nlp` module.

```
>>> import nlp;
```

The module provides methods for performing nlp tasks.

8.1. Tokenization

Methods are provided for performing tokenization at different levels of natural language.

8.1.1. 1. word_tokenize

The `word_tokenize` method is provided for tokenizing input text at the word level.

There is a `str` variant and a `sym` variant:

- `list\<str\> word_tokenize(str text)`

```
>>> import nlp;
>>> str text = "Hello world! How are we doing today?";
>>> nlp.word_tokenize(text);
["Hello", "world", "!", "How", "are", "we", "doing", "today", "?"]
```

- `list\<sym\> word_tokenize(sym text)`

```
>>> import nlp;
>>> sym text = "Hello world! How are we doing today?";
>>> nlp.word_tokenize(text);
["Hello", "world", "!", "How", "are", "we", "doing", "today", "?"]
```

8.1.2. 2. sent_tokenize

The `sent_tokenize` method is provided for tokenizing input text at the sentence level.

There is a `str` variant and a `sym` variant:

- `list\<str\> sent_tokenize(str text)`

```
>>> import nlp;
>>> str text = "Hello world! How are we doing today?";
>>> nlp.sent_tokenize(text);
["Hello world!", "How are we doing today?"]
```

- `list\<sym\> sent_tokenize(sym text)`

```
>>> import nlp;
>>> sym text = "Hello world! How are we doing today?";
>>> nlp.sent_tokenize(text);
["Hello world!", "How are we doing today?"]
```