fck

Variables

Assignment

```
<type> <name> ((:: or :>) <value>, (<additional> (opt.)) (opt.))
```

If the assignment operator `::` is used, the value that the variable has been assigned to will not be returned. If the assignment operator used is `:>` then the assigned value will be returned. See the example below:

```
>>> log int example :: 3
None
>>> log int example :> 3
3
>>>
```

```
<type> <name> :: <logic test> ? <value if true> : <value if false>
```

Variables can be simply initialised without a value, or with a value, top one. for example, `float example_variable` or `float example_variable :: 17.6`. Logic tests can also be used to determine the variable assignment. The `<value if true>` and `<value if false>` values can include `<additional>` if you want to, or this can be included outside of the logic test

Types and access

- `int`

  ```
  int example
  int example :: 3
  ```

- `float`

  ```
  float example
  float example :: 5
  float example :: 0.039
  ```

  If the given value to be assigned is an `int`, then the `int` is converted into a `float`

- `bool`

  ```
  bool example
  bool example :: 0
  bool example :: true
  bool example :: 5
  ```

  Bools can be set to numerical values which are then converted into bools. Numbers less than or equal to 0 are converted into `false` and everything else is converted into `true`. If the value is a `string`, then it is converted into true unless the string is completely empty. If the value is a `list`, `vector`, or `dict`, then unless the value has a length of 0, it is converted into true.

- `string`

  ```
  string example
  string example :: "hello world"
  ```

- `sfloat`

  ```
  sfloat example
  sfloat example :: 210
  sfloat example :: 103.02, 5
  sfloat example :: , 2
  ```

  Since `sfloat` variables can take two values, the value to store and the number of significant figures, when assigning a value to it, the following are all options:

- No value, no significant figures This just initialises the variable with a default value of 0 and significant figures of 1.
- Value, no significant figures This sets the value to the given value, and calculates the minimum number of significant figures the number was given to. For example, 935.30 would be 4 significant figures. This calculated value is then assigned to the significant figures value.
- No value, significant figures This results in the value being initialised to 0, and the number of significant figures saved. When a value is next assigned to the variable, it is rounded to the previously given number of significant figures, and assigns this rounded value to the variable.

```
>>> sfloat example :: , 2
>>> example :: 537
>>> log example
540
>>>
```

- Value, significant figures The value is rounded to the given number of significant figures, which is then saved as the value, and the given significant figures is then saved.

- `list`

```
list example
list example :: [1, 3.5, "example"]
```

Lists contain multiple values of possibly different types. They can be multidimensional

- `vector`

```
vector example
vector example :: [1, 3, 5.3]
vector example :: [15, -31, 12.5], int
vector example :: , string
```

Similar to `sfloat` variables, `vector` variables take two values for assignment, the value and the value type. `vector` variables are similar to `list` variables, with the added requirement that all items in the `vector` variable are of the same type. The assignment can take the following options:

- No value, no item type The vector is initialised with an empty vector and no item type. When the first value is then added to the empty vector, the type of the item added is determined and this is set to the item type of the list.
- Value, no item type The type of the items in the value vector is determined and this is assigned to the vector type.
- No value, item type This will initialise the vector value with an empty vector, and set the vector item type to the given item type.
- Value, item type This will set the vector value to the given value, and then set the item type to the given type. If this does not match the calculated value for the given value, an error is raised.

To calculate the type of the given value of the vector, the type of each given item is first determined. These are then compared. The types must all be the same or be similar, otherwise an error is raised. If the item types include one of either `int`, `float`, or `sfloat` values, then the value with the highest priority will be chosen, and all values will be converted into this highest priority. The priority list is as follows:

1. `float`
2. `sfloat`
3. `int`

For example, if the value vector contains `int` and `float` values, the item type is set to `float` and the `int` values are converted into `float` values

- `dict`

```
dict example
dict example :: ["key": 2, "second key": "value"]
```

The dict is similar to a list, but each value has a key associated with it. This key can be an int, float, or string, and the value can be of any type. To access a value two methods can be used:

1. Access with key To get the value associated with a given key, the following syntax can be used:

```
example[<key>]
```

If the key does not exist in the dictionary, an error is raised.

2. Access with index To access a value by index in the dictionary, the following syntax can be used:

```
example.values()[<index>]
```

The built-in method `values()` for the dictionary class returns a list of the values in the dictionary.

Operations

- `+`

Add

```
<int, float, sfloat> + <int, float, sfloat>
```

For example:

```
>>> 1 + 2
3
>>>
```

The `+` symbol can also be used with a single value, however this has no effect:

```
>>> +(-3 + 1)
-2
>>>
```

- `-`

Minus

```
<int, float, sfloat> - <int, float, sfloat>
```

For example:

```
>>> 5 - 3
2
>>>
```

The `-` symbol can also be used with a single value to give the negative of the value:

```
>>> -(19 - 3)
-16
>>>
```

- `*`

Multiply

```
<int, float, sfloat> * <int, float, sfloat>
```

For example:

```
>>> 2 * -3
-6
>>>
```

- `/`

Divide

```
<int, float, sfloat> / <int, float, sfloat>
```

For example:

```
>>> 12 / 4
3.0
>>>
```

The second value cannot be zero. If the second value is a zero, literally or as the result of evaluating an expression, a divide by zero error is raised.

- `//`

Floor division

```
<int, float, sfloat> // <int, float, sfloat>
```

For example:

```
 >>> 12 // 5
2
>>>
```

The floor division method returns the floor of the division method. Just as the division method cannot divide by zero, the floor division method cannot floor divide by zero. This also results in a divide by zero error is raised.

- `%`

  Modulo

  ```
  <int, float, sfloat> % <int, float, sfloat>
  ```

  For example:

  ```
   >>> 17 % 2
  1
  >>>
  ```

  Returns the remainder of dividing the two values. The second value cannot be a zero. A value modulo zero raises a divide by zero error.

- `**`

  Power

  ```
  <int, float, sfloat> ** <int, float, sfloat>
  ```

  For example:

  ```
   >>> 2 ** 4
  16
  >>>
  ```

- `:+:` and `:+>`

  Variable addition

  ```
  <(int, float, or sfloat) variable identifier> (:+: or :+>) <int, float, sfloat>
  ```

  The assignment operator `:+:` adds the given value onto the given variable and does not return anything. The assignment operator `:+>` does the same as `:+:` but return the newly assigned value of the variable.

  ```
   >>> float example :: 3.14159
  >>> example :+: 2
  >>> log example
  5.14159
  >>>
  ```

  ```
   >>> int example :: 19
  >>> log example :+> 6
  25
  >>>
  ```

- `:-:` and `:->`

  Variable addition

  ```
  <(int, float, or sfloat) variable identifier> (:-: or :+>) <int, float, sfloat>
  ```

  The assignment operator `:-:` subtracts the given value onto the given variable and does not return anything. The assignment operator `:->` does the same as `:-:` but return the newly assigned value of the variable.

  ```
   >>> sfloat example :: 5020, 3
  >>> example :-: 300
  >>> log example
  5320
  >>>
  ```

```
>>> float example :: 0.00539
>>> log example :-> 0.0001
0.00529
>>>
```

- `:*:` and `:*>`

  Variable multiplication

  ```
  <(int, float, or sfloat) variable identifier> (:*: or :*>) <int, float, sfloat>
  ```

  The assignment operator `:*:` multiplies the given value onto the given variable and does not return anything. The assignment operator `:*>` does the same as `:*:` but return the newly assigned value of the variable.

  ```
  >>> int example :: 21
  >>> example :*: 5
  >>> log example
  105
  >>>
  ```

  ```
  >>> float example :: 31.2
  >>> log example :*> 2
  62.4
  >>>
  ```

- `:/:` and `:/>`

  Variable division

  ```
  <(int, float, or sfloat) variable identifier> (:/: or :/>) <int, float, sfloat>
  ```

  The assignment operator `:/:` divides the given variable by the given value and does not return anything. As with normal division, the given value cannot evaluate to a zero. If it does, this returns a divide by zero error. The assignment operator `:/>` does the same as `:/:` but return the newly assigned value of the variable.

  ```
  >>> float example :: 9
  >>> example :/: 3
  >>> log example
  3.0
  >>>
  ```

  ```
  >>> sfloat example :: 0.005
  >>> log example :/> 0.0001
  50
  >>>
  ```

- `++`

  Variable positive iteration

  ```
  <(int, float, or sfloat) variable identifier> ++
  ++ <(int, float, or sfloat) variable identifier>
  ```

  The operator `++` increments the given variable by 1. The order in which this is done on its own makes no difference to the outcome, but if used in conjunction with a variable assignment, the order does make a difference. If the `++` operator is used after the variable identifier, then the increment is added after the variable assignment:

  ```
  >>> int example_1 :: 5
  >>> int example_2 :: example_1 ++
  >>> log example_1
  6
  >>> log example_2
  5
  >>>
  ```

  If the `++` operator is used before the variable, then this is carried out before the variable assignment:

```
 >>> int example_1 :: 5
>>> int example_2 :: ++ example_1
>>> log example_1
6
>>> log example_2
6
>>>
```

- `--`

  Variable negative iteration

  ```
   <(int, float, or sfloat) variable identifier> ++
  ++ <(int, float, or sfloat) variable identifier>
  ```

  The operator `--` increments the given variable by -1. The order in which this is done on its own makes no difference to the outcome, but if used in conjunction with a variable assignment, the order does make a difference. If the `--` operator is used after the variable identifier, then the increment is subtracted after the variable assignment:

  ```
   >>> int example_1 :: 12
  >>> int example_2 :: example_1 --
  >>> log example_1
  11
  >>> log example_2
  12
  >>>
  ```

  If the `--` operator is used before the variable, then this is carried out before the variable assignment:

  ```
   >>> int example_1 :: 12
  >>> int example_2 :: -- example_1
  >>> log example_1
  11
  >>> log example_2
  11
  >>>
  ```

Built-in methods

While loop

```
while <condition> {
    <suite>
}
```

The while loop runs a suite whilst a check is true. After the suite has been run, the check is recalculated, and if the result is true, the suite is re-run. The condition does not have to return a bool. The result of the condition is converted into a bool as described in the bool variable type description.

```
 >>> int conditon_int :: -3
>>> while condition_int {
...   log condition_int
...   condition_int ++
... }
-3
-2
-1
0
>>>
```

Repeat loop

Range based

```
iterate (<starting value> to (opt.)) <ending value> (step <step value> (opt.)) ( :: <iteration variable> (opt.)) {
    <suite>
}
```

```
iterate 10 {
    <suite>
}
iterate 2 to 0 {
    <suite>
}
iterate 19 to 1 step -9 {
    <suite>
}
iterate -2 to 12 :: i {
    <suite>
}
```

The iterate function iterates over a range of values. The values iterated over depend on the values given to the method. The following are the optional formats of the iterate function:

- The range of values If only one value is given, this is the maximum value, and the starting value is set to the default of 0. If two values are given, they must be separated by `to`, for example `3 to 7`. The first given value is the beginning value, and the second is the ending value. The values given to iterate between must be int or float values.
- Step value If no step value is given, the step is either 1 or -1. If the beginning value is lower than the end value, the step value is set to 1. If the beginning value is larger than the end value, the step value is -1 If a step value is given, it must be preceded by the keyword `step`, and the given value must allow the iteration to end. If the beginning value is lower than the end value, the given step value must be positive, and if the beginning value is larger than the end value, the given step value must be negative. The step value cannot be 0
- Iteration variable During the iteration loop, the current value being iterated over can be accessed during iteration. To do this, the last part of the iteration method should look similar to a variable assignment. For example `iterate 3 to 5 step 0.2 :: i` would iterate over the range from 3 to 5 with a step of 0.2, where each iteration, the current iteration value is saved to the variable `i`. The type of the variable is determined by the step and beginning value. If both are int values, then the iteration variable is also an int. Otherwise the iteration variable is a float.

The iteration iterates over the given values with the given step from the beginning value up to and including the ending value. For example `iterate -1 to 3` would iterate over the values -1, 0, 1, 2, 3.

during the iteration, if an iteration value has been assigned, this value can be altered during iteration. See the following example:

```
>>> iterate 1 to 10 step 2 :: i {
...   if i == 3 {
...     i :: 8
...   }
...   log i
...   }
1
8
10
>>>
```

If the iteration value is altered during iteration, the step value is preserved.

List or Vector based

```
iterate <list or vector> (step <step value> (opt.)) ( :: <iteration variable> (opt.)) {
    <suite>
}
```

The iteration can also be based on a list or vector. This iterates over the values in the list or vector. If a step value is given, the iteration steps through the list or vector with the given step value. The step value must be a positive non-zero int. The step value is defaulted to 1 if not given

```
>>> vector iterable_vector :: [0, 5, 3, 9], int
>>> iterate iterable_vector[1:] step 2 :: i {
...   log i
... }
5
9
>>> iterate iterable_vector step 2 :: i {
...   log i
... }
0
3
>>>
```

If, elif, and else statements

```
if <check> {
    <suite>
}
elif <check> {
    <suite>
}
else {
    <suite>
}
```

The if, elif, and else statements are simple checks. The checks can either be single or multiple checks.

Functions

```
<return type(s)> <function name> ((<variable type> <variable name> (:: <default value> (opt.)))*) {
    <suite>
}
```

```
int factorial (int base) {
    int res :: 1
    iterate 1 to base :: i {
        res :*: i
    }
    return res
}
```

```
<int, float, sfloat> + <int, float, sfloat>
```

For example:

```
>>> 1 + 2
3
>>>
```