The Remix Programming Language

An introductory programming language

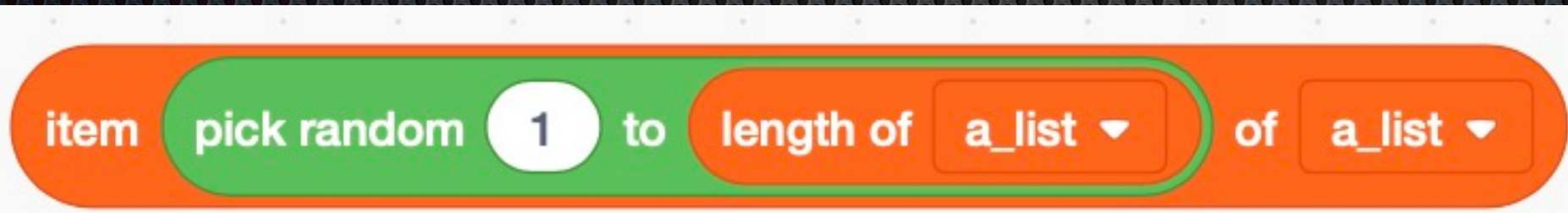
Robert Sheehan

Ideas

- A language to help with program design
- Readable
- Self documenting
- Avoid common problems
- None of these things are entirely possible

Also inspired by block programming languages

- One of the great things about block languages is that the verbal syntax is very readable.



- The same thing in Python is:

```
random.choice(a_list)
```

  or

```
a_list[random.randrange(len(a_list))]
```

- The same thing in Remix is:

```
choose from 'a list'
```

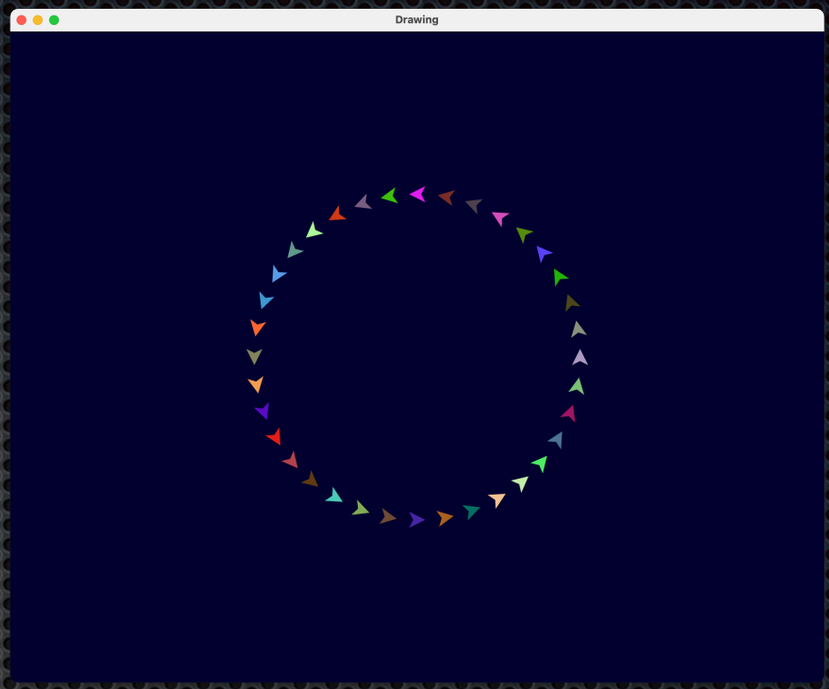
  or

```
'a list' { random ('a list' length) }
```

Developing a program

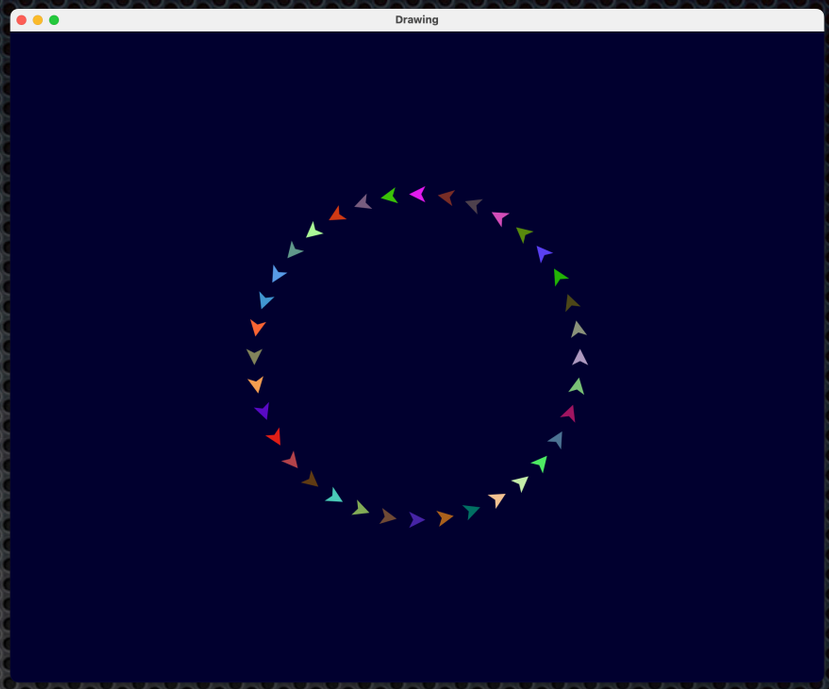
- The next sequence of slides shows how a programmer can develop a program starting with a pseudocode description of the program.
- At each step there is a direct translation into Remix as the programmer refines the code.
- Parameters get generalised when appropriate.

Draw 36 circling arrows



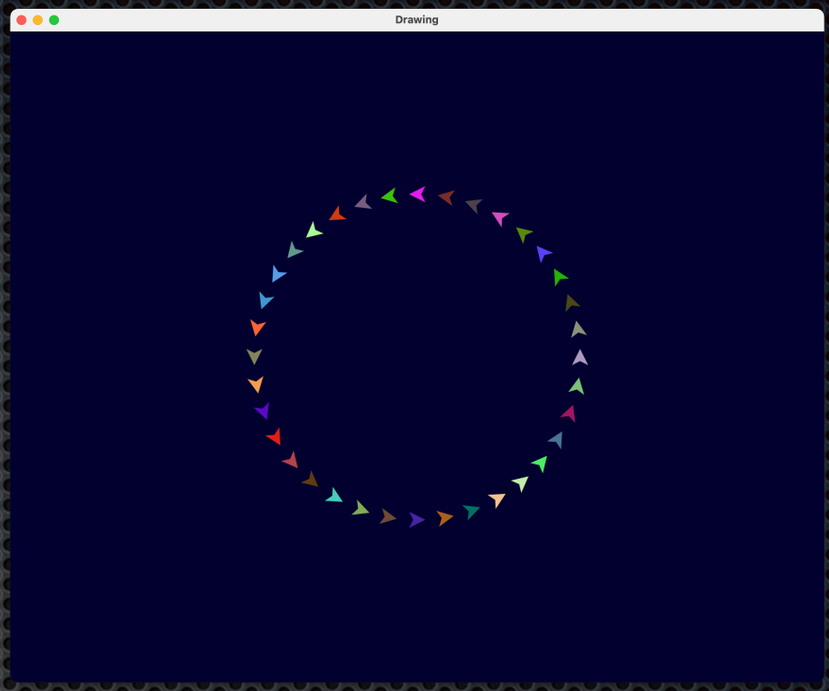

```
================
Rotating Arrows
================

prepare the window

prepare the arrows

animate the arrows
```

Draw 36 circling arrows




```
================
Rotating Arrows
================

prepare the window

'the arrows' : 36 arrows

animate 'the arrows'
```

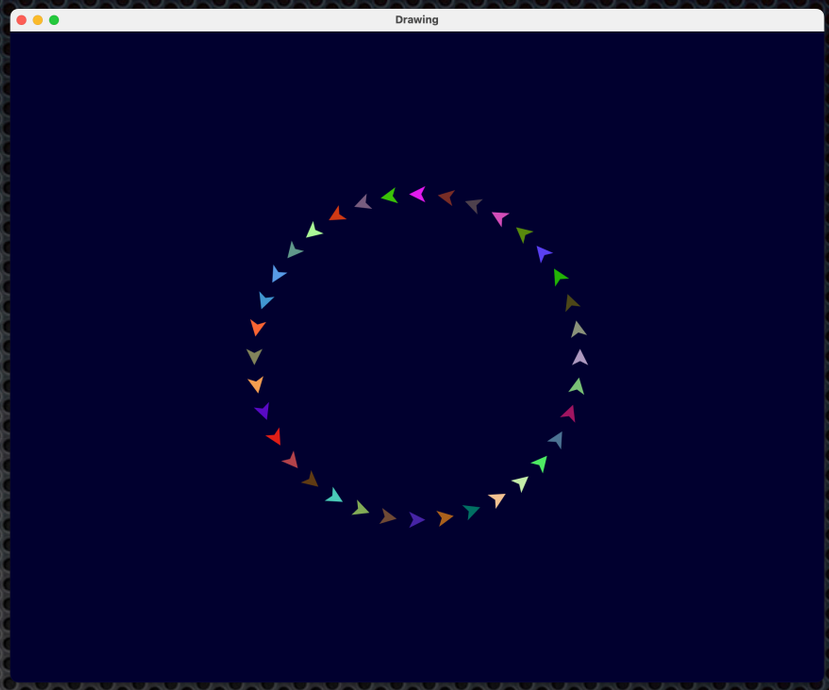
Draw 36 circling arrows




```
================
Rotating Arrows
----------------

prepare the window

'the arrows' : 36 arrows

animate 'the arrows'
```

```
36 arrows :
    make a list for the arrows
    repeat 36 times
        create a new arrow
        position the arrow
        add the arrow to the list
    return the list
```

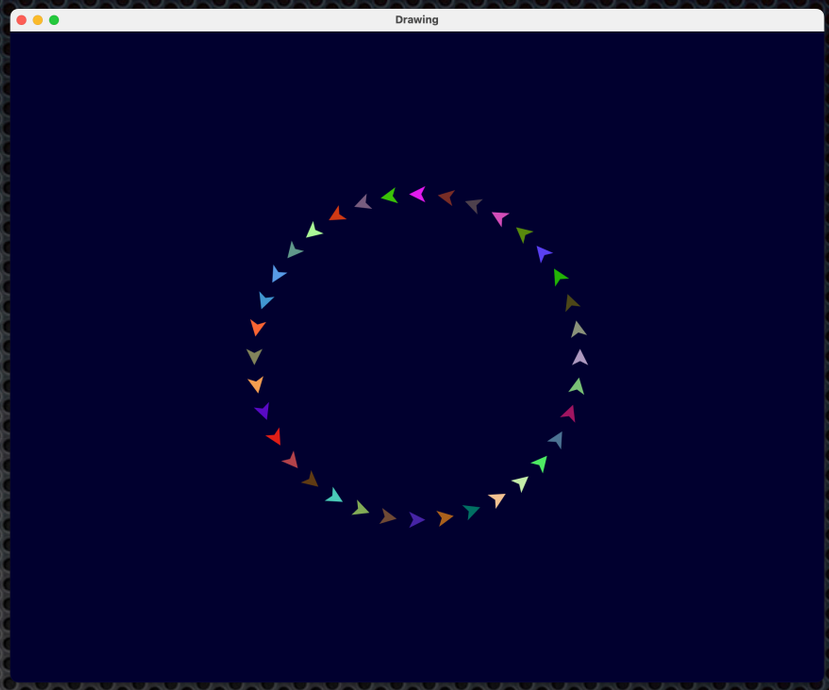
Draw N circling arrows




```
================
Rotating Arrows
----------------

prepare the window

'the arrows' : 36 arrows

animate 'the arrows'
```

```
'N' arrows :
    make a list for the arrows
    repeat 'N' times
        create a new arrow
        position the arrow
        add the arrow to the list
    return the list
```

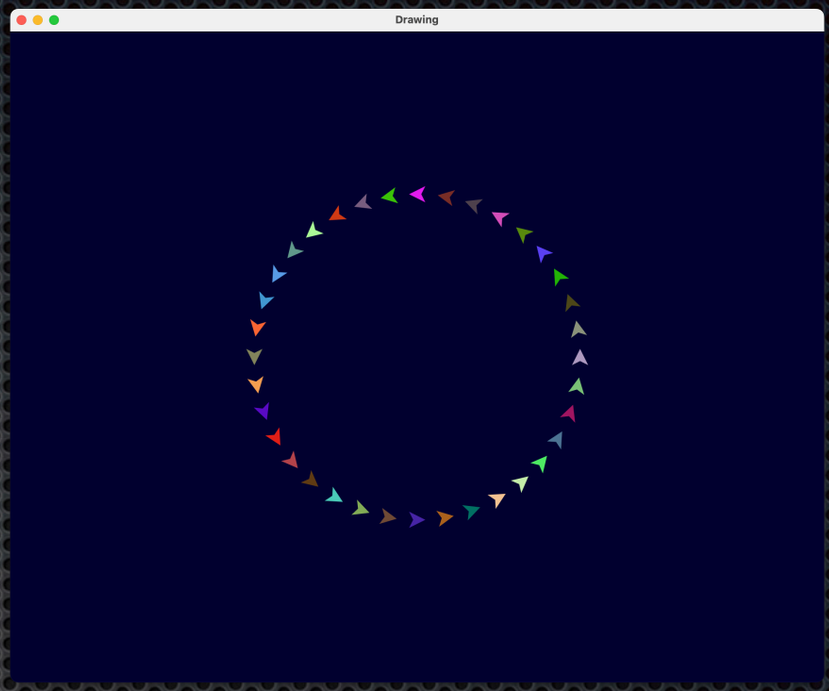
Create an arrow




```
================
Rotating Arrows
----------------

prepare the window

'the arrows' : 36 arrows

animate 'the arrows'
```

```
'N' arrows :
    make a list for the arrows
    repeat 'N' times
        'arrow' : an arrow
        position the arrow
        add the arrow to the list
    return the list

an arrow :
    make an arrow shape
    set the arrow colour
```

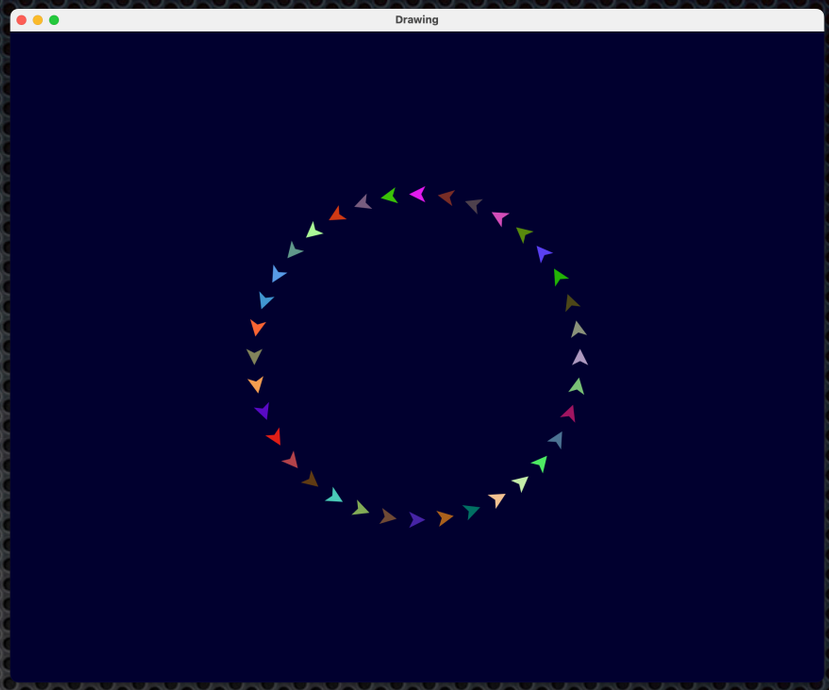
The arrow shape and colour




```
================
Rotating Arrows
----------------

prepare the window

'the arrows' : 36 arrows

animate 'the arrows'
```

```
'N' arrows :
    make a list for the arrows
    repeat 'N' times
        'arrow' : an arrow
        position the arrow
        add the arrow to the list
    return the list

an arrow :
    'arrow' : a shape from {
        { 0, 1 }
        {-2, 2 }
        { 0,-2 }
        { 2, 2 }
    }
    'arrow'’s 'colour' : a random colour
    return 'arrow'
```

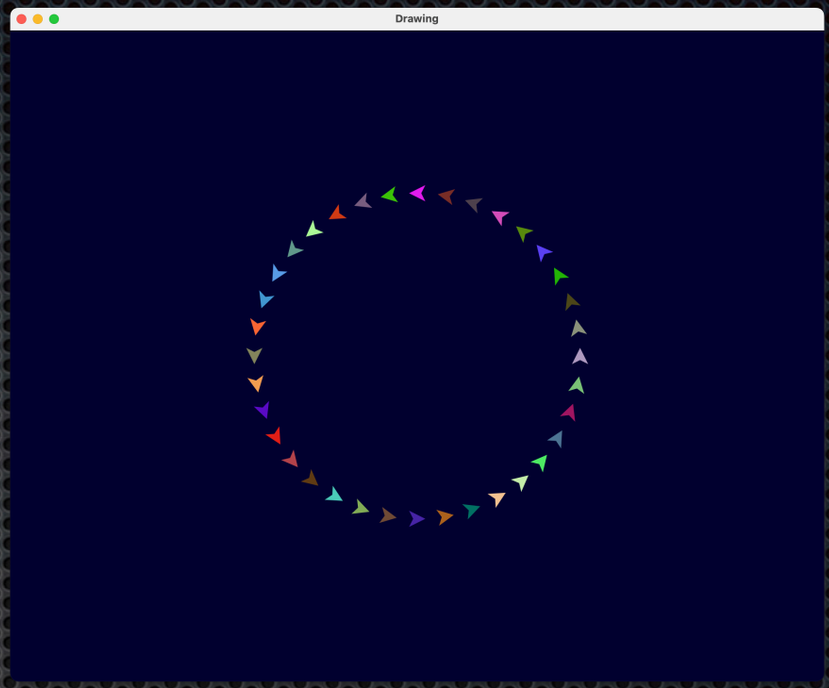
The arrow list




```
================
Rotating Arrows
================

prepare the window

'the arrows' : 36 arrows

animate 'the arrows'
```

```
'N' arrows :
    'arrows' : { }
    repeat 'N' times
        'arrow' : an arrow
        position the arrow
        append 'arrow to 'arrows'
    return 'arrows'

an arrow :
    'arrow' : a shape from {
        { 0, 1 }
        {-2, 2 }
        { 0,-2 }
        { 2, 2 }
    }
    'arrow'’s 'colour' : a random colour
    return 'arrow'
```

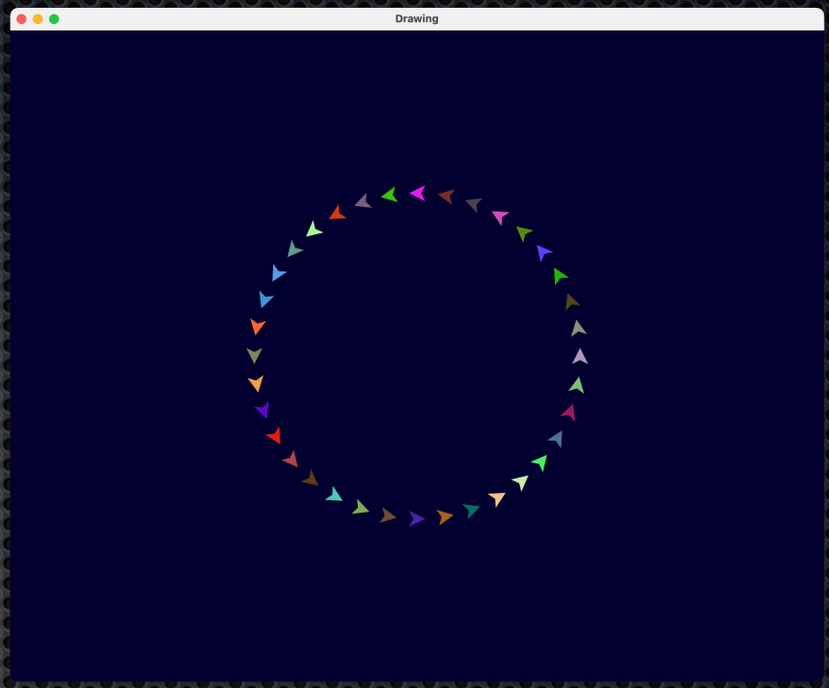
Positioning the arrows




```
'N' arrows :
'arrows' : { }

repeat 'N' times
    'arrow' : an arrow
    position 'arrow'

    append 'arrow' to 'arrows'
return 'arrows'
```

```
position 'the arrow' :
    'x' : CENTRE-X + 200 × cosine ('angle')
    'y' : CENTRE-Y − 200 × sine ('angle')
    'the arrow''s 'position' : { 'x', 'y' }
    'the arrow''s 'heading' : −'angle'
```

What about the angle?




```
'N' arrows :
    'arrows' : { }
    'angle' : 0
    repeat 'N' times
        'arrow' : an arrow
        position 'arrow' at 'angle'
        add 10 to 'angle'
        append 'arrow' to 'arrows'
    return 'arrows'
```

```
position 'the arrow' at 'angle' :
    'x' : CENTRE-X + 200 × cosine ('angle')
    'y' : CENTRE-Y – 200 × sine ('angle')
    'the arrow'’s 'position' : { 'x', 'y' }
    'the arrow'’s 'heading' : –'angle'
```

We need to animate the arrows

```
================
Rotating Arrows
================

prepare the window

'the arrows' : 36 arrows
place 'the arrows' in the window
animate 'the arrows'
```

The window is the locus of the animation.
We can create animation layers in the window.
Before we can do any animation we need to create the window and place the arrows in the window.

Preparing the window for animation

```
================
Rotating Arrows
================

'the window' : a graphics window
'the animation layer' : 'the window' next layer
'the arrows' : 36 arrows
place 'the arrows' in 'the animation layer'
animate 'the arrows'
```

Animating the arrows

```
===============
Rotating Arrows
===============

'the window' : a graphics window
'the animation layer' : 'the window' next layer
'the arrows' : 36 arrows
place 'the arrows' in 'the animation layer'
animate 'the arrows'
```

```
animate 'the arrows' :

    for each 'arrow' in 'the arrows'
        get the angle
        change the angle
        position the arrow at angle
```

Animating the arrows

```
===============
Rotating Arrows
===============

'the window' : a graphics window
'the animation layer' : 'the window' next layer
'the arrows' : 36 arrows
place 'the arrows' in 'the animation layer'
animate 'the arrows'
```

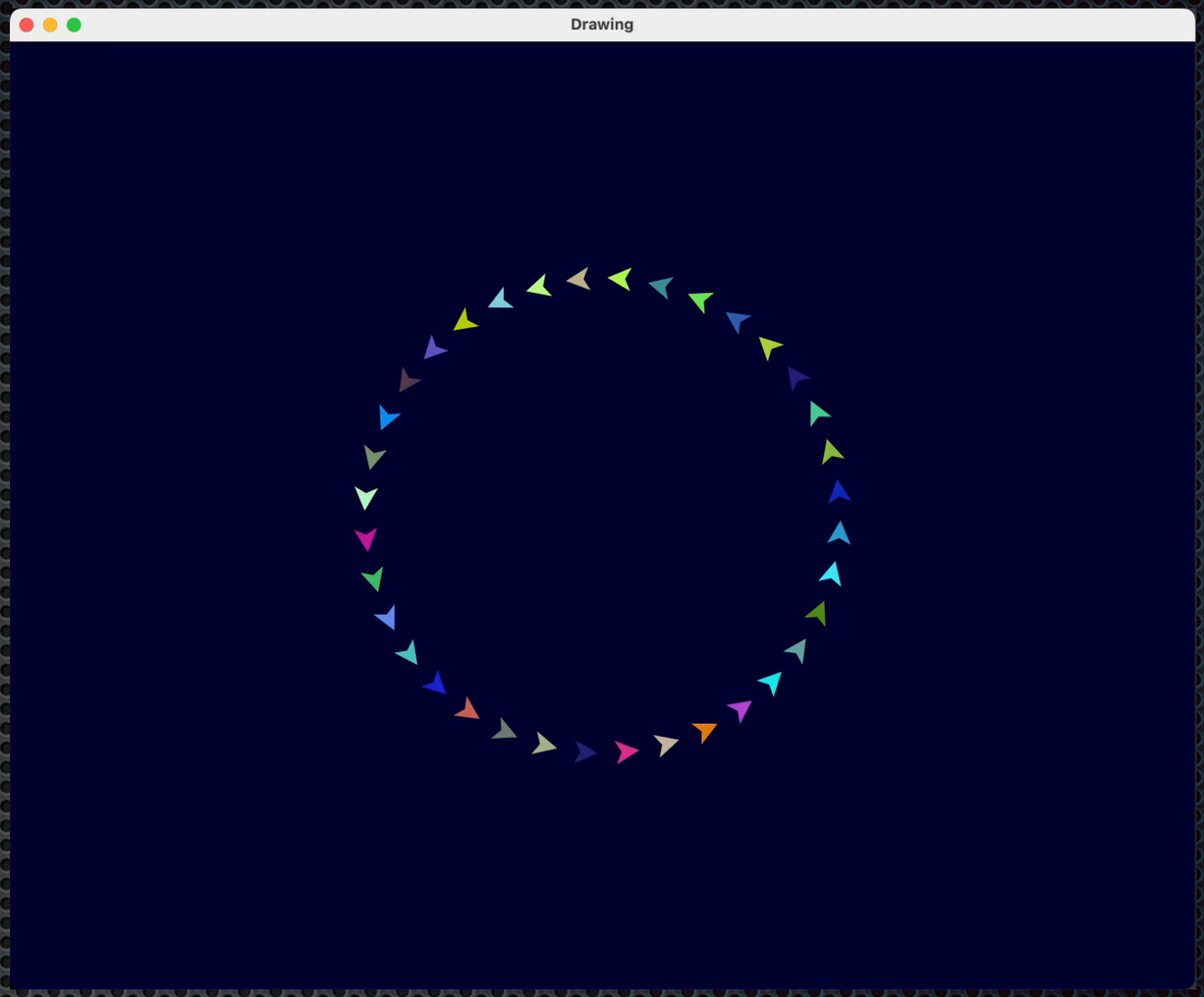
```
animate 'the arrows' :

    for each 'arrow' in 'the arrows'
        'angle' : -('arrow''s 'heading')
        add 2 to 'angle'
        position 'arrow' at 'angle'
```

Animating the arrows

```
===============
Rotating Arrows
===============

'the window' : a graphics window
'the animation layer' : 'the window' next layer
'the arrows' : 36 arrows
place 'the arrows' in 'the animation layer'
animate 'the arrows' in 'the window'
```

```
animate 'the arrows' in 'the window' :
    animate 'the window' 180 times at 30 ticks per second
        for each 'arrow' in 'the arrows'
            'angle' : -('arrow''s 'heading')
            add 2 to 'angle'
            position 'arrow' at 'angle'
```

```
using graphics lib

    ================
    Rotating arrows
    ================

    'the window' : the graphics panel
    'the animation layer' : 'the window' next layer
    'arrows' : 36 arrows
    place 'arrows' in 'the animation layer'
    animate 'arrows' in 'the window'

'n' arrows :
    'arrows' : { }
    'angle' : 0
    repeat 'n' times
        'the arrow' : an arrow
        position 'the arrow' at 'angle'
        add 10 to 'angle'
        append 'the arrow' to 'arrows'

an arrow:
    'arrow' : a shape from {
        { 0, 1 }
        {-2, 2 }
        { 0,-2 }
        { 2, 2 }
    }
    'arrow''s 'colour' : a random colour
    'arrow'

position 'the arrow' at 'angle' :
    'x' : CENTRE-X + 200 × cosine ('angle' degrees as radians)
    'y' : CENTRE-Y - 200 × sine ('angle' degrees as radians)
    'the arrow''s 'position' : { 'x', 'y' }
    'the arrow''s 'heading' : -'angle' degrees as radians

animate 'arrows' in 'the window' :
    animate 'the window' 180 times at 30 ticks per second
        for each 'arrow' in 'arrows'
            'angle' : -('arrow''s 'heading') radians as degrees
            add 2 to 'angle'
            position 'arrow' at 'angle'
```

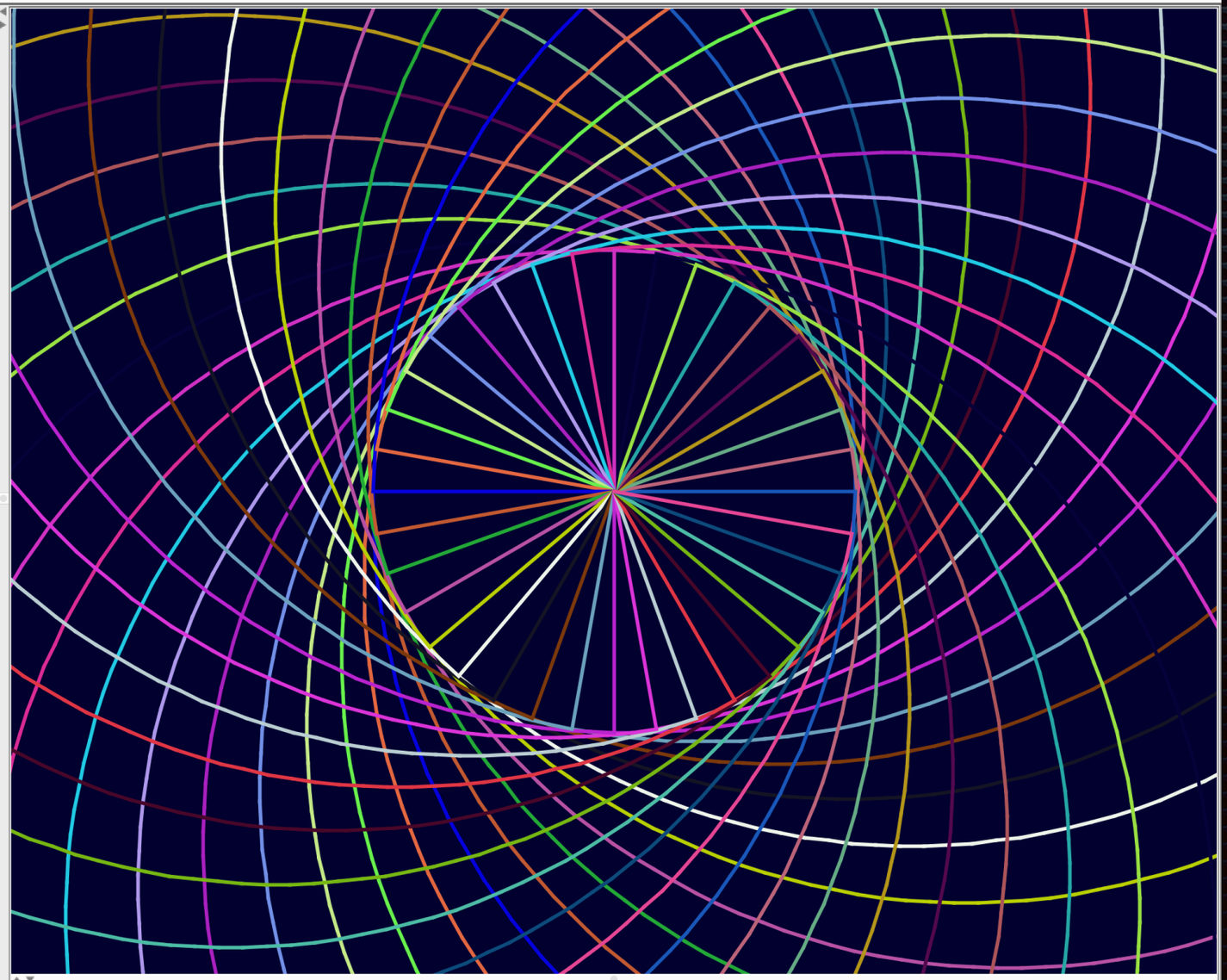


A similar turtle graphics program - with turtle objects

Remix

File Edit Control

```
with turtle graphics

    ===============
    Rotating Arrows
    ===============

    'space' : turtle space in (the graphics panel)
    'arrows' : a group of 36 turtles
    place 'arrows' group in 'space'
    prepare 'arrows'
    position 'arrows'
    repeat 180 times
        animate 'arrows'

prepare 'arrows' :
    hide 'arrows'
    'angle' : 0
    for each 'arrow' in 'arrows'
        'arrow''s 'colour' : a random colour
        'arrow''s 'heading' : 'angle'
        add 10 to 'angle'

position 'arrows' :
    move 'arrows' 200 steps
    turn 'arrows' -90 degrees

animate 'arrows' :
    move 'arrows' 20 steps
    turn 'arrows' -2 degrees
```



line: 28, offset: 17, style: default

Other examples

- Factorial
- The rainfall problem
- Fizz-buzz
- Insertion sort

Other examples

- Factorial
- The rainfall problem
- Fizz-buzz
- Insertion sort

– The beautiful factorial program –

```
'n'! :
    if ('n' = 0)
        return 1
    'n' × ('n' – 1)!

for each 'n' from 1 to 10
    'n', "! = ", 'n'! ↲
```

Other examples

- Factorial
- The rainfall problem
- Fizz-buzz
- Insertion sort

```
======================
The rainfall problem.
======================
'rain fall' : { -1, -3, 0, 5, -2, 1.0, 0, -1, 6, 7, 999, -2, 0, 10 }

'valid data' : keep 'x' from 'rain fall' where ['x' ≥ 0] until ['x' = 999]

if (no 'valid data')
    "No valid data." ↲
… otherwise
    "The average rainfall is ", average of 'valid data' ↲
```

Other examples

- Factorial
- The rainfall problem
- Fizz-buzz

```
==========
Fizz-Buzz
==========
```

- Insertion sort

```
for each 'number' from 1 to 100 do
    when
        ['number' is divisible by 15] do ["fizz-buzz" _]
        ['number' is divisible by 3 ] do ["fizz" _]
        ['number' is divisible by 5 ] do ["buzz" _]
        'number' _
```

Other examples

- Factorial
- The rainfall problem
- Fizz-buzz
- Insertion sort

```
================
 Insertion Sort
================

'list' : { -1, -3, 0, 5, -2, 1.0, 0, -1, 6, 7, 999 }

insertion sort 'list'

insertion sort 'list' :
    for each 'position' from 2 to ('list' length) do
        for each 'this' from 'position' to 2 do
            swap if element to the left is greater than 'this' in 'list'

swap if element to the left is greater than 'this' in 'list' ::
    'left' : 'this' - 1
    if ('list' {'left'} > 'list' {'this'})
        ('list' {'left'}) swap ('list' {'this'})
    … otherwise
        return
```

N.B. transparent short circuit "::"

One with tests

```
=========================
Find the middle number.
=========================

middle of 'n1' 'n2' 'n3' :
    when
        ['n1' between 'n2' and 'n3'] do ['n1']
        ['n2' between 'n1' and 'n3'] do ['n2']
        'n3'

'n1' between 'n2' and 'n3' :
    (('n2' ≤ 'n1') and ('n1' ≤ 'n3')) or (('n3' ≤ 'n1') and ('n1' ≤ 'n2'))
-------------------------------------------------------------------------
test {
    "Left"
        middle of 2 1 3
    … expected 2,
    "Centre"
        middle of 1 2 3
    … expected 2,
    "Right"
        middle of 1 3 2
    … expected 2
}
```

The Remix Language - keywords

- `true`
- `false`
- `null`
- `return`
- `redo`
- `create`
- `extend`
- `getter / setter`
- `ME / MY`
- `↲ / ~`
- `library`
- `using`

The Remix Language - function names

- A collection of *words* and *parameters*
- A *word* is a sequence of Unicode characters surrounded by whitespace or separator characters
  e.g. `! what a* and item √ list ++`
  `(a few characters not allowed)`
- A *parameter* is an identifier 'any text within single quotes'
  - *parameters* can be inside parentheses ( ) or brackets [ ]
  - there are reference parameters which start with #
- Any sequence of *words* and *parameters* is a function name
- Should be on one line (when defining)
- Ends with : (or sometimes ::)

Example function names

```
'value1' and 'value2' :
not at end of 'list' :
repeat ['block'] until ['condition'] :
for each #item from 'start' to 'finish' do ['block'] :
filter 'list' by #item where ['condition'] :
draw 'colour' circle of 'size' at 'centre' :
'number' is not divisible by any in 'list' :
'N' ! :
#number ++ :
```

When defining a function we represent blocks with [name] - for the documentation system

The Remix Language - function calls

- a function call is like a function name but with all of the formal parameters replaced with expressions evaluating to the actual parameters (and no terminating ":"s)
- Lisp problem of too many parentheses
- Don’t require parentheses if unambiguous
  - separated by
    - operator, brace, bracket, comma, colon
    - but also literal string, numeric or logic values
    - and identifiers (variable names)
- Can only access local variables and parameters and constants (no global variables)
  - but there are reference parameters, and composite types (list, string, map, object) can be modified in function calls

Not like Python

What is the output here?

```
def print_a():
    print(A)

def change_and_print_a():
    A = 4
    print_a()
    print(A)

A = 3
print_a()
print(A)
change_and_print_a()
print(A)
```

Answer:
3
3
3
4
3

What is the output here?

```
print a :
    'A' ↲

change and print a :
    'A' : 4
    print a
    'A' ↲

'A' : 3
print a
'A' ↲
change and print a
'A' ↲
```

Answer:
null
3
null
4
3

'A' is null

The Remix Language - syntax

- Generally every line is one statement
  - But a line continues to following lines if brackets/braces/ parentheses are not closed
  - Also an ellipsis … at the start of a line indicates it is a continuation
  - And a "." can be used to terminate a statement allowing multiple statement lines
- Indentation is meaningful (even more than in Python)

The Remix Language - indentation

- Code starting at the left side margin is top-level code
  - function definitions always start in the top-level (except in libraries)
    - all function definitions are gathered before any other statements are run (in the main program and when running libraries)
    - you can put your functions at the top or bottom of your source file (or anywhere in-between)
  - all other statements at the top-level are executed in order

The Remix Language - indentation and deferred blocks

- Tabs are used to indent code
- Most indented areas of code are *deferred* blocks. They contain statements which will be evaluated at some later time.
- Blank lines are ignored (the indentation of following lines is still important).

This is really important

- We can pass blocks of code as parameters to functions in a syntactically trivial manner
- e.g. in the following function call there are two blocks passed as parameters, one explicit, one implicit

```
starting with [ 'size' : 100 ] repeat 4 times
    plot 'arrow' around centre with 'size'
    'size' : 'size' + 100
```

Comments

- Comments are flexible
  - A line where the first visible character is "-"
  - Multiline comments have a "=" at the start of a line and continue until a closing "=" at the start of another line
  - Any ";" outside a string continuing to the end of the line

```
=== A comment start ===
    Inside the comment.
=== The comment end ===

    - lists can be comments
    - like this

'a' : 5 ; this is a comment too
```

Statements

- assignment statement

```
'my name' : "Robert"
```

- return statement - can return nothing

```
return something
```

- redo statement

```
redo
```

- print statement

```
"My name is ", 'my name', "." ↲
```

- expression
- using library statement

```
using graphics lib
    block of code which uses the library
```

Expressions

- arithmetic - standard mathematical precedence
- boolean expressions
- function calls - always have a value
  - if no explicit return, uses the result of the last statement, some statements have the value "null"
- accessing a list or map element

```
'rain fall' {2}, 'person' {"name"}
```

- a variable, constant or literal: number, string, logic or list/map/block/object value

Lists

```
'rain fall' : { -1, -3, 0, 5, -2, 1.0, 0, 6, 7, 999, -2, 0, 10 }
```

- Lists hold any collection of types - even blocks

```
{['ball' {'x'} < 'size'] ,['ball' {'x'} > STD-WIDTH - 'size']}
```

- They expand as required
- Accessed with `{index}` - from 1

```
'rain fall' {3} or 'rain fall' {'N'}
```

Ranges

```
1 to 10, 5 to -5
```

- direction implicit
- ranges (and lists) have built-in iterators
  - only one per range/list currently
- iterator functions (also work on lists, strings, maps and sequence objects)

```
'iterator' : start 'range'
next 'iterator'
end 'iterator'
```

Maps

- A map/hash/dictionary - with keys and values
- keys are strings
- The { } are used for lists and maps.
  - If the elements are
    - key : value
  - then Remix creates a map, otherwise it creates a list.

```
'a shape' : {
    "position" : { 0, 0 }
    "heading"  : 0 ; straight up
    "size"     : 5
    "colour"   : BLACK
    "points"   : { }
}

from 'shape' with 'points' :
    'new shape' : copy 'shape'
    'new shape' {"points"} : 'points'
    'new shape'

'b shape' : from 'a shape' with { { 1, 2 } }
```

Objects

- Objectives
  - make the creation and use of objects as simple as possible
  - keep it consistent with the rest of the language
- The inverse of Uniform Function Call Syntax
  - instead of allowing function calls to use the method calling syntax, make method calls use the function call syntax
  - at one level this is as simple as passing the object explicitly as a parameter, but this parameter can be anywhere in the parameter list

Objects

- The *create* function makes objects with fields and methods.
- A constructor is an ordinary function call that returns an object.

```
a person called 'name' :
    create
        'name' : 'name'

    getter
        'name'
```

- In a method signature the ME or MY parameter represents the object reference.
  - Can call methods on the same object from a method with ME referring to this object.
- Parameters can be the same as field names e.g. 'name'.

```
say hello to ME :
    "Hi ", 'name' ↲

welcome ME and bye :
    say hello to ME
    "bye now." ↲

'person' : a person called "Mary"
'person''s 'name' ↲
say hello to 'person'
welcome 'person' and bye
```

Getters and Setters

- object fields are private
- automatic getters and setters can be specified using *getter*, *setter* or *getter/setter* blocks
- a getter is called by:

```
'the object''s 'field'
```

- a setter is called by:

```
'the object''s 'field' : new value
```

- both getters and setters can be defined manually if access control is desired
  - manually defined getter function

```
MY 'field' :
```

  - manually defined setter function

```
MY 'field' 'value' :
```

Extending

- Remix does not have inheritance
- but you can extend existing objects using a similar syntax to object creation
  - *extend (existing-Object)* instead of *create*
  - this creates a new object based on the existing one
- *extend* can add new fields and methods
- it can also replace existing method definitions
- in this way you can specialise objects in a way similar to inheritance

```
a person called 'name' :
    create
        'name' : 'name'

        MY introduction :
            "My name is ", 'name', "." ↲

        say hello to ME :
            "Hi ", 'name', ", pleased to meet you." ↲

a person called 'name' age 'years':
    extend a person called 'name'
        'age' : 'years'

        say hello to ME :
            "Hi ", 'name', ", you are ", 'age', " years old." ↲

'Mary' : a person called "Mary"
'Mary' introduction
say hello to 'Mary'

'Jack' : a person called "Jack" age 30
'Jack' introduction
say hello to 'Jack'

say hello to (a person called "Stu" age 56)
```

Output

```
My name is Mary.
Hi Mary, pleased to meet you.
My name is Jack.
Hi Jack, you are 30 years old.
Hi Stu, you are 56 years old.
```

Blocks and Block Sequences

- Blocks hold code for later execution - deferred
  - the basis for control structures
- Both explicit `[ ... ]` and implicit with indentation

- A block sequence is a block with one or more statements (or lines)
  - the main program and all function/method bodies are block sequences
- Looping is done by calling `redo` - this takes control back to the start of the block sequence
  - this is pretty low-level and not intended for beginner use

Example control structure

```
– While a "condition" is true keep doing "block"...
while ['condition'] repeat ['block'] :
    if (not (do 'condition'))
        return 'last'
    'last' : do 'block'
    redo

– ... and using it
'x' : 2
'result' : while
    'x' is divisible by 2
… repeat
    add 3 to 'x'
'result' ↲
```

Output: 5

Transparent functions

- Because a block is effectively an anonymous function, return inside a block doesn’t always mean a return from its enclosing function.
- We can make this happen using "transparent" functions
  - this is only used once in the current standard library and should never be necessary in novice code
  - we declare a transparent function by ending with "::"
    - see the insertion sort example earlier
  - if you call a transparent function from the top-level this returns back to the level above the top-level which terminates the program

Reference functions

- or reference parameters - the parameter name starts with "#"
- Another consequence of creating control structures in Remix itself
- e.g. in a for loop function we need to pass a reference to the index or current value so that it can be accessed in the body of the for loop - which is just a block parameter

```
for each #item in 'list' ['block'] :
    'pos' : start 'list'
    do
    if (end 'pos')
        return 'last'
    #item : next 'pos'
    'last' : do 'block'
    redo
```

Another control structure

With a reference parameter

```
–Create a new list by applying a block to each element of a list.
–Uses the variable name passed as #Item.
apply 'block' to each #item from 'list' :
    'result' : { }
    for each #item in 'list'
        append (do 'block') to 'result'
    'result'

–... and using it
'result' : apply ['a' × 'a'] to each 'a' from (1 to 10)
'result' ↲
```

```
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

The Remix Language - built-in functions and standard lib functions

- Built-in functions include
  - mathematical operations
  - if/do/type/output functions
  - list/range/map functions
- Graphics and turtle graphics
  - graphics library has both Remix and Java functions
  - turtle graphics library implemented solely in Remix using the graphics library
- Control structures (except if/otherwise)
  - are all written in Remix (can be inspected and changed by the user)

Current implementation

- Built in Java using ANTLR grammar
- Currently an interpreter
- A simple IDE for the language
  - helps with code layout
  - basic function completions
  - built in graphics and text output

Some research questions

- Is it helpful or just a bother?
- Assuming it is helpful:
  - Does Remix encourage students to write pseudocode or plan their code?
  - What is the best level to transition natural language statements into "code" like statements?
  - What is the best way to convey information about parameter requirements in the function name?
  - What style of function name is most readable - e.g. should prepositions and articles be included? (Code in Remix written by a speaker of English contains lots of prepositions in function names.)
  - How applicable is this approach to programmers from different languages, Te Reo?
    - function call structure is completely flexible
  - Does Remix make plagiarism checking simpler/more reliable?