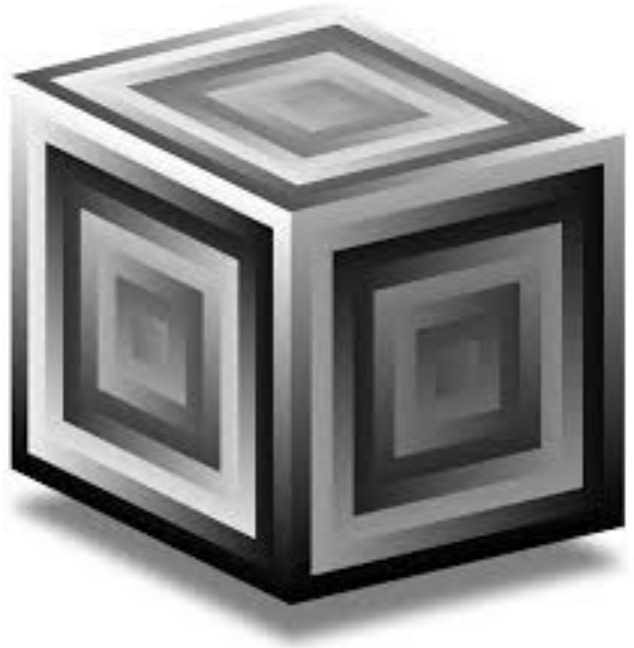
# sclang Basics

# Topics Addressed

- Overview of sclang
- Types of data
- Operator precedence
- Functions
- Conditionals
- Loops
- Arrays



# sclang

- sclang is the computer language we use to control/make our sounds but it does not actually produce the sounds we hear (this is the role of scsynth)
- sclang is a language that can send OSC (Open Sounds Control) messages to scsynth where the sound is created. To summarize: sclang -> OSC -> scsynth -> audio
- Not all code written in sclang sends OSC messages to scsynth, only that which is intended to produce sound.
- sclang can be used as a regular old programming language just like Java or Python.
- sclang has many similarities to Python. This will make it easier to learn!

# What's different from Python?

- All statements in sclang must end in a semicolon. This is similar to other languages like Java or C.
- To run code, you should highlight the block of code you want to run and hit SHIFT – RETURN (MacOS). Alternatively, you can go to menu at the top and select "Language" and then "Evaluate File". Also see the docs about "smart" code evaluation using COMMAND – RETURN.
- Parentheses are used to define a region for the execution of a block of code. More on this shortly.
- The last expression evaluated gets posted to the post window with "->" preceding it.
- There are many other things as well. I will try to point them out as we go.

# Numbers

Like Python, there are two types of numbers: integers and floats.

- Floats are decimal point numbers
- Standard operators like +, -, *, / apply
- / is floating point division regardless if the operands are integers
- Need to use `.div()` for integer division. Ex: `3.div(2)`

```
3; // This is an integer
3.14; // This is a float
```

Inline comments are denoted with a double backslash. Multiline comments can be started with /* and closed off with */.

# Operator Precedence

- This is important: There is no operator precedence in SuperCollider!
- What then is the result of this expression?
  - 3 + 4 * 7?
  - The answer: 49!
- Binary operators like + or * are evaluated from left to right in all cases.
- Therefore, you should make use of parentheses to ensure your code evaluates as you express.
  - How do we fix the line above?
  - 3 + (4 * 7)

# Booleans

- SuperCollider uses the keywords `true` and `false` (lowercase)
- Python's `and` is &&
- Python's `or` is ||
- Python's `not` is also `not` but is used differently

Short-circuiting does occur if the second operand is a function and the result of the function does not affect the Boolean result.

```
true; // This is the Boolean true
false; // This is the Boolean false
true.not; // Note how not is used here. This is because not is a method.
          // More on methods later.
(3 < 4) || (3.1 > -1); // Note that here I use parentheses to force the
                       // bools to evaluate first.  This is true.
(3 < 4) && (3.1 < -1); // This evaluates to false.
```

# Exercise: What does these evaluate to?

| Expression | Result |
|---|---|
| `3 * 4 - 2` | `10` |
| `4 - 2 < 3 - 4` | Error. `4 - 2 < 3` evaluates to `true`. We can't do `true - 4`. |
| `3 < 4 \|\| 2 - 3 < 1` | Error. We will attempt to evaluate `false \|\| 2`. |
| `3 < 4 \|\| (2 - 3 < 1)` | `true` |
| `3 < 4 \|\| (1 < 2 - 3)` | Error. We will attempt to do `true - 3`. |

Lesson: Use parentheses judiciously!

# Strings and Symbols

- SuperCollider has two text types: Strings and Symbols
- Strings are demarcated with double quotes
- Symbols are demarcated with single quotes
- Both are sequences of characters
- Difference (from the docs): “Unlike strings, two symbols with exactly the same characters will be the exact same object[1]. Symbols are optimized for recreating the same symbol over and over again. In practice, this means that symbols are best used for identifiers or tags that are only meaningful within your program, whereas you should use a string when your characters are really processed as text data. Use symbols to name things, use strings for input and output.”

# Strings and Symbols Examples

```
"This is a string"; // A string (i.e., a piece of text)
'This is a symbol'; // A symbol.  Single quotes allow you to include white space
\symbol; // A symbol. With a backslash you cannot have white space

a = 'hi'; // A symbol
b = \hi; // The same symbol.  This will be the same thing in memory.

a == b; // Do they have the same value?  True
a === b; // Are they the same thing in memory? True

c = "hi"; // A string
d = "hi"; // A string with the same value

c == d; // Do they have the same value? True
c === d; // Are they the same thing in memory? False
```

# Global Variables

- Interpreter variables
  - These are the letters a-z.  They can be used at any place in a program.
  - They are global variables
  - You should avoid them in larger programs because it is easy to lose track of what has been assigned to them and what they are.
  - NEVER use **s** as a global variable.  It is reserved to communicate with scsynth.
- Environment variables
  - For all intents and purposes of this class, these are also global variables (though not actually)
  - Each program has one or more environments that can store variables that can be accessed throughout the program.
  - To create an environment variable, use a tilde before the name

# Global variables

```
a = 3; // Assigns the interpreter variable a to 3
b = 4; // Assigns the interpreter variable b to 4
a + b; // Displays 7 in the post window

~envVar = 3; // Creates an environment variable and assigns it to 3
currentEnvironment; // This displays all the environment variables in the post window
                    // The post window will show: -> Environment[(envVar -> 3)]
                    // You can use this variable throughout your program

a + ~envVar; // Displays 6 in the post window

~coolVar = 12; // Creates an environment variable and assigns it to 12
currentEnvironment; // Post window shows: -> Environment[(envVar -> 3), (coolVar -> 12)]

~envVar * ~coolVar; // Displays 36 in the post window
```

# Parentheses and Local Variables

- Parentheses can be used in expected contexts like precedence in mathematical expressions.  Ex. (3 + 4) * 5
- Parentheses can be used to articulate a block of code to be evaluated in sequential order.
  - Using parentheses in this context creates a scope where local variables can be defined.
  - Work done inside this scope does not persist after evaluation unless interpreter or environment variables were assigned.
  - One advantage of using parentheses is it makes evaluation quicker through “smart” evaluation (COMMAND-RETURN)
  - Local variables can be declared with the keyword var

# Local variables with Parentheses

```
// Clicking anywhere in the lines and using ”smart” evaluation will evaluate the three
// lines in order without having to highlight them.

(
a = 4; // Sets the interpreter variable a to 4
b = 3; // Sets the interpreter variable b to 3
a + b; // Evaluates the expression a + b.
)

a; // Even though a and b were assigned within the scope above, we can still access them because
   // these are interpreter variables

(
var test = 4; // Define a local variable test.  It is only valid within the parentheses.
test.postln; // Print the variable to the post window.
)
test; // We cannot access a local variable outside of its scope.  Error!
```

# Functions

- Functions are vitally important in SuperCollider
- We say that SuperCollider supports first-class functions
  - This simply means functions are treated as values just like integers or strings. We can assign them to variables and pass them into other functions.
  - Python is like this as well
- Functions can have parameters. We use the keyword arg or | | (vertical lines) to specify a parameter
- There is no return keyword. The last expression evaluated in a function is returned.
  - This makes things like early return tricky (still doable)
- All functions return a value. If no expressions are used in the function, then the keyword `nil` is returned.
- Functions are declared with {} brackets

# Comparing Python and SuperCollider Functions

Name

Parameter

```
def square(x):
    return x * x
```

Value returned

Name

Parameter

Value returned

```
~square = {|x| x * x};
```

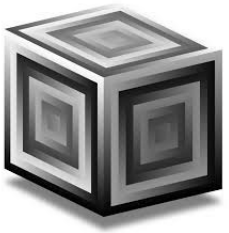
Name

Parameter

Value returned

```
~square = {arg x; x * x};
```

Note that when you use the keyword arg that you must follow the parameter declaration with a semicolon



# Evaluating functions

- To evaluate a function, we use the `.value` method.
  - Note that we will define methods more specifically later when we discuss classes.
- Here are two ways to evaluate the square function we defined on the previously slide.

```
~square.value(4);
```

Name of function  Argument

```
value(~square, 4);
```

Name of function  Argument

# Splitting functions across lines

```
~coolFunction = {arg a; var x = a * a; x - 2};
```

OR

```
~coolFunction = {
	arg a; // argument
	var x = a * a; // local variable x
	x - 2 // Returns value of last expression
};
```

# Conditionals

- Conditionals are different from Python
- The if branch and the else branch must be functions!
- Similar to Python an expression whose resulting value is a Boolean determines whether the if or else branch is taken.
- Unlike Python, the if syntax is an *expression* and not a statement. This means that we can assign an if statement to a variable!

```
if (<Boolean expression>, {true function}, {false function})
```

# Conditionals and Functions

```
a = 24; // An interpreter variable a

b = if(a < 4, {100}, {200}); // The if expression will return the last expression of the false func.

b; // This will be assigned to 200


~inHumanRange = {
    arg freq; // Parameter
    if ((freq >= 20) && (freq <= 20000), // && is the equivalent of the keyword ”and” in Python
        {(freq.asString + "is in human range").postln;}, // True function
        {(freq.asString + "is not in human range").postln;} // False function
    )
};


~inHumanRange.value(40); // This will post “40 is in human range” to the post window.

// Question: All functions return values?  What will ~inHumanRange.value(40) return?

// Answer: A function always returns the last expression evaluated.  In this case, that will be “freq // + “is in human range””
```

# Exercise: write a function `~isInAMajor`

Write a function called `~isInAMajor` that takes a symbol representing a note and posts whether or not the note is in A major and returns true or false.

```
~isInAMajor = {
    |noteSymbol|
    var inAMajor = if ((noteSymbol == 'A') || (noteSymbol == 'C#') || (noteSymbol == 'E'),
        {
            (noteSymbol.asString + "is in A major").postln; // Need .asString to concatenate
            true
        }, {
            (noteSymbol.asString + "is not in A major").postln;
            false
        }
    );
    inAMajor
};
```

# Exercise: write a recursive function for factorial

```
~factorial = {
    |x|
    if (x == 0, {1}, {x * ~factorial.value(x - 1)});
};
```

# Scope

- What is scope?
  - Refers to the visibility of variables.
  - To learn more, take a programming languages class!
  - Key insight below:

```
(
var num = 4;
var func = {num}; // A function has access to its outer context
func.value.postln;
)
```

This posts the number 4!

# Loops

While loops: `while({<test function>}, {<body function>})`

EX:

```
// Post the numbers 0 through 3
i = 0;
while({i < 4}, {i.postln; i = i + 1});
```

For loops: `for(startValue, endValue, {<body function>})`

EX:

```
// Post the numbers 3 through 6
for(3, 6, {arg x; x.postln;});
```

- the end value of the for loop is inclusive.
- You can use the value iterated over in the for loop and pass it in as an argument to the body function.
- Use a `forBy` loop for a variable step. Standard for loop increments by one.

# What does the following code do?

```
for(0, -1, {arg i; i.postln;});
```

Be careful! This does not post nothing. It posts the numbers 0 and -1.

```
for(0, 0, {arg i; i.postln;});
```

Be careful! This does not post nothing. It posts the number 0.

# Exercise: write an exercise to print out the partials of a starting frequency up to a max.

Given a starting frequency of 440Hz, the partials would be 440Hz, 880Hz, 1320Hz, 1760Hz…

```
~freqMax = 20000;
~startFreq = 440; // Concert A
~currentFreq = ~startFreq;
while(
	{~currentFreq < ~freqMax},
	{
		~currentFreq.postln;
		~currentFreq = ~currentFreq + ~startFreq;
	}
);
```

# Arrays and Lists

- In Python, if we want to store an ordered collection of objects, we use a list.
- SuperCollider also has lists which are equivalent to Python lists. It is more common though to use arrays.
- Arrays in SuperCollider are equivalent to lists except that they have a fixed maximum size beyond which they cannot grow.
  - They too are a collection of heterogenous types

# Arrays and Lists

- Arrays are vectors that store data up to a maximum size.
- Use these for fixed size storage.  Can be more efficient.
- In truth, if you try and add values beyond the maximum size, SuperCollider will let you and return to you a new array.

- Lists are resizable Arrays.
- Equivalent to Python.
- Lists are more flexible, but have more overhead
- No size limitation

# Relevant Array Notation

```
a = Array.new(45); // Create an array of max. size 45
a.postln; // Shows array on post window. What gets displayed? Answer: []
a = a.add(3) // Append the number 3 to the array. a is now [3].


a.add(3) // WARNING: The add method potentially returns a new list.  Sometimes it
modifies the original list.  Sometimes not.  This is due to how array is implemented
under the hood.  Always assign to variable as shown above.


a[0].postln; // Posts the value 3 from [3]
b = a[0]; // Assign value 3 from [3] to variable b
a = a.add(nil); // Arrays can hold different types. nil is Python's None.
a.class; // Gives back the type of variable a. In this case, Array.
c = Array.newClear(4); // Gives back an array with four slots filled
                       // with nils. [nil, nil, nil, nil]
d = [1, 2, 3]; // Create an array with initial values
d.maxSize; // Reports the max size for the array.  A power of 2.  In this case, 4.
```

# Relevant Array Notation

```
a = Array(45); // Create an array of max. size 45.  Don't need ".new"
a.maxSize; // In truth your max. size will be a power of two
a.size; // Size represents the current size.  So zero in this instance.

a[0] = 1; // Index error because size is currently zero
a = a.add(2); // Append first element to the array
a[0] = 3; // Change first element to three; Equivalent to put method
a.put(0, 4); // Change first element to four -> .put(index, element)
a.insert(0, "hi"); // Insert element "hi" at index 0 -> .insert(index, element)

b = [1, 2, 3]; // Declare an array and initialize its values to 1, 2, and 3
c = a ++ b; // Concatenate two arrays

d = #[1, 2, 3]; // Immutable array.  More efficient.
// Check documentation for all the methods you can use on arrays.
```