Table of Contents

Learn ezr^2

Why Learn ezr^2?

ezr^2 is a programming language that's **easy to learn** and **practical to use**. ezr^2 can be learnt by anyone, **of any age**, in a few minutes. Anyone can extend the functionalities of ezr^2 with **libraries**. If you already know C#, you can even help in ezr^2 development with **C# Assisted ezr^2 Libraries** (**CSAELs**)! CSAELs bring the existing functionality of C# to ezr^2! Experienced ezr^2 programmers can even ditch the boilerplate syntax for the shorter **QuickSyntax**. The normal syntax satisfies the beginner, as it is easy to use and QuickSyntax satisfies the expert, as it is very short.

Documentation

Start coding with the help of ezr^2's official documentation! Read the text in **bold** for a TL;DR. (Heavily inspired by ***this Python tutorial***)

Hello World, Displaying Text to The Screen and Strings

There's a tradition in which programming tutorials start with a so-called Hello World program. A **Hello World program** simply **prints the words "Hello world" to the screen**, and the `show()` **function lets you do that**. You will learn more about functions later.

```
show("Hello, World!")
```

The `show()` function takes the value you put between the parentheses and prints it to the screen. This value is known as the *argument*. When you want to show "Hello world" to the screen, what you're displaying is text, and **text in ezr^2 is always put between double quotes**. In the world of computer programming, **this is called a *string***.

The **double quotes around a string mark its the start and end**. This way, a string is easy to recognize for ezr^2. Here are a few more examples of valid strings:

```
"Hello world"
"My name is Uday"
"This one is a bit longer. There is no limit to how long a string can be!"
```

Nothing

After printing anything to the screen, the `show()` **function also displays "nothing"**. This means the **function `show()` returned no value, but printed the argument to the screen**. If you **write `"Example"` and run, only `"Example"` will be shown to the screen**. As in the expression `"Example"` **returned the**

string "Example". `nothing` **is a representation of a "lack of value"** - equivalent to null (C#, C, Java, etc) or None (Python).

Numbers

Just like strings, you can ask ezr² to **print numbers using the `show()` function**. Unlike strings, **numbers don't need quotes around them**. So, the code to print the number 10 would be:

```
show(10)
```

Try some of these operators on numbers!

Operator	Name	Example
+	addition	2 + 6
-	subtraction	8 - 4
*	multiplication	3 * 23
/	division	6 / 2
%	modulo	9 % 2
^	powered by	4 ^ 2

Floating-point Numbers

Try the **expression** `3 / 2`. **It returns** `1` **instead of the correct answer,** `1.5`. In computer programming, **there's a strong distinction between non-fractional numbers like 1, 3, and 42 and fractional numbers like 3.14 and 5.323. The former are called *integers* - *whole numbers*, while the latter are called *floats* - *floating-point numbers***. Now try this:

```
show(3 / 2.0)
```

This should display `1.5` to the screen, because you have specified that `2.0` is a float. **An integer divided by another integer will always return an integer, but the same done with a float will always return a float**. **This rule is the same for all operators**.

Variables

What if you want to store the result of a calculation, to be accessed later in the code? For this, you use *variables*. ***Variables* allow you to store items in memory for as long as the ezr² program runs**. It's like a reservation for an item, under the variable name, in the system memory.

The **syntax to assign a variable is so:** `NAME: VALUE`. To assign the number 42 to a variable called age, you would write:

```
age: 42
```

And just like numbers and strings, you can **print a variable with the `show()` function**. In the following example, you assign a few variables and then print them. Here you are also **adding strings together - this combines or *concatenates* the two strings**. You have to **convert the `age` variable to a string** to be able to add it to another string, so you **use the built-in `as_string` function in the integer**.

```
age: 42
name: "Joe"

show("Hello, my name is " + name)
show("My age is " + age.as_string())
```

User Input

Let's make a simple adder - the user will enter two numbers, and the adder will return the result of adding them together. To do so you need to **get the user's input**. You can **use the `get()` function** for that. You have to **feed it the message to show to the user**. You can **enter `nothing` if you don't want any message with the input request**.

The code should look something like this:

```
num1: get("Enter num 1: ")
num2: get("Enter num 2: ")

show(num1 + num2)
```

If you try the code, you will see that it's not adding the numbers, but concatenating them! As in, if you enter `2` and `4` as `num1` and `num2` respectively, the result will be `24`, not `6`.

This is because the **`get()` function returns a string.** You need to **convert the string into an integer**. To do so you **use the built-in `as_integer()` function in the string**. You'll immediately convert the results of the `get()` functions to integers. Try the below code:

```
num1: get("Enter num 1: ").as_integer()
num2: get("Enter num 2: ").as_integer()
```

```
show(num1 + num2)
```

Conditions

Try entering normal text as the input for the above code - you'll get an error! This is because the `as_integer()` **function cannot convert normal text to integers**. To avoid the user seeing the messy error, you can **use the `try_as_integer()` function**. But, **this function returns `nothing` if it is unable to convert the string to an integer**! If you try to add `nothing` and a number together you'll get an error! You have to **make sure that `num1` and `num2` are integers** before you add them. You can **do this with *if expressions***. **The if expression has a body of code that only executes whenever the if statement's condition is met**. **The additional else statement, which also has a body of code, runs if the if condition is false**. **The else if statement is used when you wish to satisfy one statement while the other is false**.

```
num1: get("Enter num 1: ").try_as_integer()
num2: get("Enter num 2: ").try_as_integer()

if num1 = nothing do
        show("Num 1 is invalid!")
else if num2 = nothing do
        show("Num 2 is invalid!")
else do
        show(num1 + num2)
end
```

Here are all the comparison operators:

Operator	Operation	Example
=	equal to	"i" = "i", "i" = "j"
!	not equal to	"i" ! "i", "i" ! "j"
>	greater than	5 > 8, 9 > 4
<	less than	5 < 8, 9 < 4
>=	greater than or equal to	5 >= 5, 5 >= 2, 5 >= 8
<=	less than or equal to	5 <= 5, 5 <= 2, 5 <= 8
or	and	3 = 2 and 2 > 1, 2 = 5 and 5 = 2, 2 = 2 and 2 > 1

Operator	Operation	Example
and	or	3 = 2 or 2 > 1, 2 = 5 or 5 = 2, 2 = 2 or 2 > 1

If you use **comparisons outside if expressions** they **will return a *boolean***. **Booleans are just two values - *true* or *false***.

One-liners

You might have noticed that the if expression has the *end* keyword at the end. **All multi-line ezr² expressions *must* end with the *end* keyword**. But **you can write most multi-line expressions in one line - these are called *one-liners***. They are the same as writing the multi-line version of an expression, but they must fit in a single line. For example -

```
if get("Hello there!\n") = "General Kenobi" do show("Nice")
```

Note that **new lines can be coded as the semicolon (`;`) symbol**. **Comments are signified by the at symbol (`@`) at the start**. Comments are just more information about the written code - like how the code works. They do not affect the execution of any code and are ignored by ezr².

Loops

For the above scripts, you might have found it annoying to have to copy and paste the code again and again to try it out. What if you want the code to run forever? Or even a set number of times? Do you have to keep copying and pasting it? No! You'll use *loops* for that. **A loop keeps executing the given body of code till a condition is satisfied.** ezr² has two types of loops - *count loops* and *while loops*.

Count Loops

Count loops* repeat the given code a *set number of times.

```
count to 10 do
        show("Hello, World!")
end
```

The **count loop can optionally keep account of the *iteration variable***. **The iteration variable, starting at zero, is the value that gets incremented each loop or *iteration***. The count loop checks if the iteration variable is less than the max iterations - if not it stops the loop. **Iteration variables are useful if you want to iterate over a *list* or *array*, using the variable as the index**.

```
count to 10 as i do
        show("The iteration variable, named 'i' is: " + i.as_string())
```

```
end
```

Now, what **if you don't want the loop / iteration variable to start at zero**? You can also **set the start of a count loop**!

```
count from -5 to 5 as i do
        show("The iteration variable, named 'i' is: " + i.as_string())
end
```

Lastly, what if you **want the iteration variable to skip a few numbers**? Say you only want even numbers? You can **set the *step* of the count loop**! The step is what the loop adds to the iteration variable every iteration.

```
count to 10 step 2 as i do
        show("The iteration variable, named 'i' is: " + i.as_string())
end

count from 1 to 10 step 2 as i do
        show("The iteration variable, named 'i' is: " + i.as_string())
end
```

While Loops

***While loops* repeat the given code till the given condition turns false, and revaluate the condition every iteration**.

```
password: get("Enter password: ")
while password ! "yeet" do
        password: get("Password is false! Try again: ")
end
```

While loops **can also be used to run code infinitely**!

```
while true do
        show("Hello, World!")
end
```

Iteration Control

What if you want to stop a loop while it's running or want to skip an iteration on a condition? You can **use the *stop* and *skip* keywords to control the iterations of loops**. **The *skip* keyword stops the**

current iteration and starts the next. **The *stop* keyword just stops the loop**. Let's code a simple example.

```
count to 10 as i do
        if i = 5 do
                skip
        end

        if i = 8 do
                stop
        end

        show("I = " + i.as_string())
end
```

In this example, the iteration is *skipped* when i = 5 and the loop is outright *stopped* when i = 8. **The *skip* and *stop* keywords can also be used in while loops**.

Arrays

If you try the one-liner version of the count loop, like -

```
count to 10 do show("Hi!")
```

You'll see a whole lot of `nothing`-s in parentheses, separated by commas! That's an *array*.

Arrays* are used to store multiple items of any type in a single variable**. **It is ordered and unchangeable, or *immutable. When you say that **arrays are ordered**, it **means that the items have a defined order, and that order will not change**. An array is **created by putting items, seperated by commas, between parenthesis**. **Array items are indexed**, the first item has index `0`, the second item has index `1` and so on. **Items of an array can be accessed with the `<` operator** -

```
array_example: (1, "string", 3.54, nothing, false)
show(array_example < 0)
show(array_example < 1)
show(array_example < 2)
show(array_example < 3)
show(array_example < 4)

show(array_example < "sd")
```

That last line of code should show an error - the **item after the `<` operator must be a valid index in the array**!

Here are all the array operators:

Operator	Name	Example
*	duplication	(2,3) * 4
/	division	(6,2,3,4) / 2
<	item access	(6,2) < 1

Now, what if you want to create an array with one item? Let's try it -

```
array_example: (3)
show(array_example)
```

This won't work, as **ezr² thinks `(3)` is part of an operation - not an array**. So, you have to **have a comma after the first item**!

```
array_example: (3,)
show(array_example)
```

You can **access the length of the array with the built-in `length` variable in array** -

```
array_example: (1,2,3,4)
show(array_example.length)
```

Lists

Lists* are like arrays, but they are changeable or *mutable. They are **created with square brackets, instead of parentheses**.

```
list_example: [1,"string",1.45,nothing,true]
show(list_example)
```

Here are all the list operators:

Operator	Name	Example
+	append	lst: [1,5]; lst + 4
-	remove	lst: [1,5]; lst - 1
*	duplication	[2,3] * 4
/	division	[6,2,3,4] / 2
<	item access	[6,2] < 1

You can **access the length of the list with the built-in `length` variable in the list**, just like with arrays.

TODO: Dictionaries, Globalization of Variables, Functions, Objects and Classes, Special Functions, Built-ins, Modules, IO and STD Libraries, QuickSyntax

C# and ezr^2

Embedding ezr^2 to your application!

ezr^2 can be embedded into other programs. This allows your users to write scripts in ezr^2 which can control various systems in your app. ezr^2 code is much simple and easier to learn and understand than, for example, C# code.

To learn more about embedding ezr^2 into your application, check the documentation ***here***

ezr^2 doesn't have [feature] built-in!

Now, this is where **CSAELs** step in. CSAELs, or, **CSharp Assisted ezr^2 Libraries** are libraries written in C# for ezr^2 scripting.

With CSAELs, anyone who knows C# can add any feature they want to ezr^2. These features can also be built-in to ezr^2 - it's a win-win situation! All you have to do is wrap C# variables, functions and whatnot with attributes the ezr^2 interpreter can identify. All these "wrapper" attributes are part of the ezr^2 API.

To learn more about CSAELs, check the documentation ***here***

TODO

TODO

Multilingual ezr²

Why?

Most programming languages like C, C#, Python, and Java, just to name a few, are English-based. Of course, they do not follow the standard grammar for natural English, but, you would agree that it would be much easier to learn these languages if you knew English. This can be a huge barrier to those who have never spoken a single word of English.

How's it different from other multilingual (programming) languages?

There are a few examples of programming languages which may not require you to know English:

Scratch Jr.

This is a programming language that is based on symbols and pictures. How far can you go with pictures?

Now, you can have language made up completely of symbols, Sign Language is a good example. But then, the user is forced learning a different language. Why not just learn English at that point and do conventional coding?

Hedy

Hedy is a transpiler - something that converts Hedy code to Python code, and executes it. There is no method in which the Python code can be converted back to Hedy code in the way Hedy code is converted into Python code. Why is this important?

Well, let's assume the role of Programmer A. They have made a project which is written in French Hedy code. They are satisfied with the project and publish it on GitHub.

Here comes Programmer B, who likes A's project and downloads it, uses it and loves it. But, they have found a few bugs in the code! Sadly, they cannot fix the code as it is written in French Hedy, and they only know Hindi Hedy.

As you can see, the language barrier has prevented the potential collaboration between two programmers.

So how *is* it different?

Imagine a programming language, which:

- Can be written in one natural language syntax;
- Can be published in a natural-language-neutral intermediate syntax (QuickSyntax);

- Can be converted back to another natural language syntax; And,
- Can be converted to QuickSyntax again to publish!

This means the natural language barrier is completely and utterly shattered! Yeah!

How?

Well, this is still in a planning stage, but the structure should look like this:

The Syntax Checkers

This includes the Lexer and Parser. There will be custom versions of these for each language, so that the translations are perfect.

The Converter

This converts the Parsed nodes from/into QuickSyntax or serialized binaries.

The Interpreter

The interpreter only interprets nodes that have been given to it. It does not know which language the source code is in.

Of course, this is quite a high-level summary, and, again, this is only in the planning stage.