

YourLanguageSucks

From Theory.org Wiki

Contents

- 1 JavaScript sucks because
 - 1.1 Poor Design
 - 1.2 Type System
 - 1.3 Bad Features
 - 1.4 Missing Features
 - 1.5 DOM
- 2 Lua sucks because
- 3 PHP sucks because
 - 3.1 Fixed in currently supported versions
- 4 Perl 5 sucks because
- 5 Python sucks because
 - 5.1 Fixed in Python 3
- 6 Ruby sucks because
- 7 Flex/ActionScript sucks because
- 8 Scripting languages suck because
- 9 C sucks because
- 10 C++ sucks because
- 11 .NET sucks because
- 12 C# sucks because
- 13 VB.NET sucks because
- 14 VBA sucks because
- 15 Objective-C sucks because
- 16 Java sucks because
 - 16.1 Syntax
 - 16.2 Model
 - 16.3 Library
 - 16.4 Disputed
- 17 Backbase sucks because
- 18 XML sucks because
- 19 XSLT/XPath sucks because
- 20 CSS sucks because
 - 20.1 Fixed in CSS3
- 21 Scala sucks because
- 22 Haskell sucks because
- 23 Clojure sucks because
- 24 Go sucks because
 - 24.1 Core language
 - 24.2 Concurrency
 - 24.3 Standard library
 - 24.4 Toolchain
 - 24.5 Community

- 25 Rust sucks because
 - 25.1 Safety
 - 25.2 API design and the type system
 - 25.3 Toolchain

JavaScript sucks because

Note some of this is not JavaScript itself, but web APIs (<https://developer.mozilla.org/en/docs/Web/API>)

Poor Design

- Every script is executed in a single global namespace that is accessible in browsers with the window object.
- Camel case sucks:

```
XMLHttpRequest  
HTMLHRElement
```

- Automatic type conversion between strings and numbers, combined with '+' overloaded to mean concatenation and addition. This creates very counterintuitive effects if you accidentally convert a number to a string:

```
var i = 1;  
// some code  
i = i + ""; // oops!  
// some more code  
i + 1; // evaluates to the String '11'  
i - 1; // evaluates to the Number 0
```

- The automatic type conversion of the + function also leads to the intuitive effect that += 1 is different than the ++ operator. This also takes place when sorting an array:

```
var j = "1";  
j++; // j becomes 2  
  
var k = "1";  
k += 1; // k becomes "11"  
  
[1,5,20,10].sort() // [1, 10, 20, 5]
```

- The var statement uses function scope rather than block scope, which is a completely unintuitive behavior. You may want to use let instead.

Type System

- JavaScript puts the world into a neat prototype hierarchy with Object at

the top. In reality values do not fit into a neat hierarchy.

- You can't inherit from Array or other builtin objects. The syntax for prototypal inheritance also tends to be very cryptic and unusual. (Fixed in ES6)
- In JavaScript, prototype-based inheritance sucks: functions set in the prototype cannot access arguments and local variables in the constructor, which means that those "public methods" cannot access "private fields". There is little or no reason for a function to be a method if it can't access private fields. (Fixed in ES6 via symbols)
- JavaScript doesn't support hashes or dictionaries. You can treat objects like them, however, Objects inherit properties from `__proto__` which causes problems. (Use `Object.create(null)` in ES5, or `Map` in ES6)
- `Arguments` is not an Array. You can convert it to one with `slice` (or `Array.from` in ES6):

```
var args = Array.prototype.slice.call(arguments);
```

(`Arguments` will be deprecated eventually)

- The number type has precision problems.

```
0.1 + 0.2 === 0.30000000000000004;
```

The problem is not the result, which is expected, but the choice of using floating point number to represent numbers, and this is a lazy language designer choice. See http://www.math.umd.edu/~jkolesar/mait613/floating_point_math.pdf.

- NaN stands for not a number, yet it is a number.

```
typeof NaN === "number"

// To make matters worse NaN doesn't equal itself
NaN !== NaN
NaN !== NaN

// This checks if x is NaN
x !== x
// This is the proper way to test
isNaN(x)
```

This is as it should be, as per IEEE754. Again, the problem is the indiscriminate choice of IEEE754 by the language designer or implementer.

- `null` is not an instance of `Object`, yet `typeof null === 'object'`

Bad Features

(You can bypass many of these bad features by using <http://www.jslint.com/>)

- JavaScript inherits many bad features from C, including switch fallthrough, and the position sensitive ++ and -- operators. See C sucks below.
- JavaScript inherits a cryptic and problematic regular expression syntax from Perl.
- Keyword 'this' is ambiguous, confusing and misleading

```
// This as a local object reference in a method
object.property = function foo() {
    return this; // This is the object that the function (method) is being attached to
}

// This as a global object
var functionVariable = function foo() {
    return this; // This is a Window
}

// This as a new object
function ExampleObject() {
    this.someNewProperty = bar; // This points to the new object
    this.confusing = true;
}

// This as a locally changing reference
function doSomething(somethingHandler, args) {
    somethingHandler.apply(this, args); // Here this will be what we 'normally' expect
    this.foo = bar; // This has been changed by the call to 'apply'
    var that = this;

    // But it gets better, because the meaning of this can change three times in a single function
    someVar.onEvent = function () {
        that.confusing = true;
        // Here, this would refer to someVar
    }
}
```

- Semi colon insertion

```
// This returns undefined
return
{
    a: 5
};
```

- Objects, and statements and labels have very similar syntax. The example above was actually returning undefined, then forming a statement. This example actually throws a syntax error.

```
// This returns undefined
return
{
    'a': 5
};
```

- Implied globals:

```
function bar() {
  // Oops I left off the var keyword, now I have a global variable
  foo = 5;
}
```

(This can be fixed by using 'use strict' in ES5.)

- The == operator allows type coercion, which is a useful feature, but not one you would want by default. The operator === fixes the problem by not type coercing, but is confusing coming from other languages.

```
0 == ""
0 == "0"
0 == " \t\r\n "
"0" == false
null == undefined

"" != "0"
false != "false"
false != undefined
false != null
```

- Typed wrappers suck:

```
new Function("x", "y", "return x + y");
new Array(1, 2, 3, 4, 5);
new Object({"a": 5});
new Boolean(true);
```

- parseInt has a really weird default behavior so you are generally forced into adding that you want your radix to be 10:

```
parseInt("72", 10);
```

You can use Number('72') to convert to a number.

- The (deprecated) with statement sucks because it is error-prone.

```
with (obj) {
  foo = 1;
  bar = 2;
}
```

- The for in statement loops through members inherited through the prototype chain, so you generally have to wrap it in a long call to object.hasOwnProperty(name), or use Object.keys(...).forEach(...)

```
for (var name in object) {
  if (object.hasOwnProperty(name)) {
    /* ... */
  }
}
```

```

    }
  }
  // Or
  Object.keys(object).forEach(function() { ... });

```

- There aren't numeric arrays, only objects with properties, and those properties are named with text strings; as a consequence, the for-in loop sucks when done on pseudo-numeric arrays because the iterating variable is actually a string, not a number (this makes integer additions difficult as you have to manually `parseInt` the iterating variable at each iteration).

```

var n = 0;
for (var i in [3, 'hello world', false, 1.5]) {
  i = parseInt(i); // output is wrong without this cumbersome line
  alert(i + n);
}
// Or
[3, 'hello world', false, 1.5].map(Number).forEach(function() { alert(i + n) });

```

- There are also many deprecated features (see https://developer.mozilla.org/en/JavaScript/Reference/Deprecated_Features) such as `getYear` and `setYear` on `Date` objects.

Missing Features

- It has taken till ES6 to enforce immutability. This statement doesn't work for JavaScript's most important datatype: objects, and you have to use `Object.freeze(...)`.

```

// It works okay for numbers and strings
const pi = 3.14159265358;
const msg = "Hello World";

// It doesn't work for objects
const bar = {"a": 5, "b": 6};
const foo = [1, 2, 3, 4, 5];

// You also can't easily make your parameters constant
const func = function() {
  const x = arguments[0], y = arguments[1];

  return x + y;
};

```

- There should be a more convenient means of writing functions that includes implicit return, especially when you are using functional programming features such as `map`, `filter`, and `reduce`. (ES6 fixed this)

```

ES6
x -> x * x

```

- Considering the importance of exponentiation in mathematics, `Math.pow` should really be an infix operator such as `**` rather than a function. (Fixed in ES6 with `**`)

```
Math.pow(7, 2); // 49
```

- The standard library is non-existent. This results in browsers downloading hundreds of KBs of code per page-hit of every webpage in the world just to be able to do things we usually take for granted.

DOM

- Browser incompatibilities between Firefox, Internet Explorer, Opera, Google Chrome, Safari, Konqueror, etc make dealing with the DOM a pain.
- If you have an event handler that calls `alert()`, it always cancels the event, regardless of whether you want to cancel the event or not

```
// This event handler lets the event propagate
function doNothingWithEvent(event) {
    return true;
}

// This event handler cancels propagation
function doNothingWithEvent(event) {
    alert('screwing everything up');
    return true;
}
```

Lua sucks because

- Variable declaration is global by default, and looks exactly like assignment.

```
do
    local realVar = "foo"
    real_var = "bar" -- Oops
end
print(realVar, real_var) -- nil, "bar"
```

- a dereference on a non-existing key returns `nil` instead of an error. This, coupled with the above point makes misspellings hazardous, and mostly silent, in Lua.
- If a vararg is in the middle of a list of arguments only the first argument gets counted.

```
local function fn() return "bar", "baz" end
print("foo", fn()) -- foo bar baz
print("foo", fn(), "qux") -- foo bar qux
```

- you can hold only one vararg at a time(in ...)
- You can't store varargs for later.
- You can't iterate over varargs.
- You can't mutate varargs directly.
- You can pack varargs into tables to do these things, but then you have to

worry about escaping the nil values, which are valid in varargs but signal the end of tables, like `\0` in C strings.

- Table indexes start at one in array literals, and in the standard library. You can use 0-based indexing, but then you can't use either of those things.
- `break`, `do while` (`while (something) do` and `repeat something until something`), and `goto` exist, but not `continue`. Bizzare.
- Statements are distinct from expressions, and expressions cannot exist outside of statements:

```
>2+2
  stdin:1: unexpected symbol near '2'
>return 2+2
  4
```

- Lua's default string library provides only a subset of regular expressions, that is itself incompatible with the usual PCRE regexes.
- No default way to copy a table. You can write a function for that which will work till you'll want to copy a table with `__index` metamethod.
- No way to impose constraints on function arguments. 'Safe' Lua functions are a mess of type-checking code.
- Lack of object model. Not bad by itself, but it leads to some inconsistencies - the string type can be treated like object, assigned a metatable and string values called with methods. The same is not true for any other type.

```
>("string"):upper()
  STRING
>({1,2,3}):concat()
  stdin:1: attempt to call method 'concat' (a nil value)
>(3.14):floor()
  stdin:1: attempt to index a number value
```

PHP sucks because

- `'0'`, `0`, and `0.0` are false, but `'0.0'` is true
- Because it's a fractal of bad design (<http://me.veekun.com/blog/2012/04/09/php-a-fractal-of-bad-design/>) and it makes me sad (<http://phpsadness.com/>).
- There is no one consistent idea of what an expression is. There are at least three: normal, plus the following exceptions:
 - here doc syntax `"<<<END"` can not be used in the initiation of method attribute default values on PHP < 5.3
- The documentation is not versioned. There is a single version of the docs that you are supposed to use for `php4.x`, `php5`, `php5.1...`
- There is no general concept of an identifier. Some identifiers (like variable names) are case sensitive, others case insensitive (like function calls):

```
$x = Array();
$y = array();
$x == $y; # is true
```



```
$x = 1;
$X = 1;
$x == $X; # is true
```

- if you mis-type the name of a built-in constant, a warning is generated and it is interpreted as a string "nonexistent_constant_name". Script execution is not stopped.
- if you send too many arguments to a user-defined function call, no error is raised; the extra arguments are ignored.
 - This is intentional behavior for functions that can accept variable numbers of arguments (see `func_get_args()`).
- if you send the wrong number of arguments to a built-in function call, an error is raised; the extra arguments aren't ignored such as with a normal function call
- `Array()` is a hash & array in one data type
 - "hash & array" would be relatively ok. *ordered* hash & array, now, that's pure evil. Consider:

```
$arr[2] = 2;
$arr[1] = 1;
$arr[0] = 0;
foreach ($arr as $elem) { echo "$elem "; } // prints "2 1 0" !!
```

- it doesn't have dynamic scoping
- it has identifier auto-vivification with no equivalent of "use strict"
- in addition to implementing POSIX STRFTIME(3), roll-your-own date (<http://us2.php.net/manual/en/function.date.php>) formatting language.
- weakass string interpolation resolver:

```
error_log("Frobulating $net->ipv4->ip");
Frobulating Object id #5->ip

$foo = $net->ipv4;
error_log("Frobulating $foo->ip");
Frobulating 192.168.1.1
```

However, this will work as expected:

```
error_log("Frobulating {$net->ipv4->ip}");
```

- there are two ways to begin end-of-line comments: `//` and `#`
- code must always be inserted between `<?php` and `?>` tags, even if it's not HTML generation code *disputed on discussion page*
- two names for the same floating point type: `float` and `double`
- there are *pseudo-types* to define parameters that accept multiple types, but no way to set a type for a specific parameter other than for objects, arrays or callables since PHP 5.4 (fixed in PHP 7)
- an integer operation overflow automatically converts the type to float
- There are thousands of functions. If you have to deal with arrays, strings, databases, etc., you'll get with tens of functions such as `array_diff`, `array_reverse`, etc. Operators are inconsistent; you can only merge arrays

with +, for example (- doesn't work). Methods? No way: `$a.diff($b)`, `$a.reverse()` don't exist.

- PHP is a language based on C and Perl, which is not inherently object oriented. And if you know the internals for objects, this actually makes you happy.
- function names aren't homogeneous: `array_reverse` and `shuffle` both operates on arrays
 - And some functions are needle, haystack while others are haystack, needle.
- string characters can be accessed through both brackets and curly brackets
- variable variables introduce ambiguities with arrays: `$$a[1]` must be resolved as `${$a[1]}` or `/${$a}[1]` if you want to use `$a[1]` or `aa` as the variable to reference to.
 - Variable Variables in general are a great evil. If variable variables are the answer, you are surely asking the wrong question.
- constants can only be defined for scalar values: booleans, integers, resources, floats and strings (they can hold arrays as of PHP 7)
- constants do not use `$` prefix, like variables - which makes sense, since they're constants and not variables.
- `!` has a greater precedence over `=` but... not in this `if (!$a = foo()) "special" case!`
- on 32 and 64 bit systems shift operators (`<<` `>>` `<=>` `>=>`) have different results for more than 32 shifts
- (instances of) built-in classes can be compared, but not user-defined ones
- arrays can be "uncomparable"
- `and` and `or` operators do the same work of `&&` and `||`, but just with different precedence
- there are both curly brackets and `:` followed by `endif;`, `endwhile;`, `endfor;`, or `endforeach` to delimit blocks for the respective statements
- there are both `(int)` and `(integer)` to cast to integers, `(bool)` and `(boolean)` to cast to booleans, and `(float)`, `(double)`, and `(real)` to cast to floats
- casting from float to integer can lead to undefined results if the float is beyond the boundaries of integer
 - This produces a fatal error on PHP5 - unless you are using an object that implements array functionality, giving you something that works as an object AND an array (and gives headaches).
- included file inherits the variable scope of the line on which the `include` occurs, but functions and classes defined in the included file have global scope
 - Class and function definitions are always global scope.
- if an included file must return a value but the file cannot be included, `include` statement returns `FALSE` and only generates a warning
 - If a file is required, one must use `require()`.
- functions and classes defined within other functions or classes have global scope: they can be called outside the original scope
- defaults in functions should be on the right side of any non-default arguments, but you can define them everywhere leading to unexpected behaviours:

```
function makeyogurt($type = "acidophilus", $flavour)
{
    return "Making a bowl of $type $flavour.\n";
}

echo makeyogurt("raspberry"); // prints "Making a bowl of raspberry ". Only a warning will be gener
```

- (PHP 4) methods can be called both as instance methods (with `$this` special variable defined) and class methods (with `self`)
- if the class of an object which must be unserializing isn't defined when calling `unserialize()`, an instance of `__PHP_Incomplete_Class` will be created instead, losing any reference to the original class
- `foreach` applied to an object iterates through its variables by default
- static references to the current class like `self::` or `__CLASS__` are resolved using the class in which the function is defined (solvable in PHP >= 5.3 with `static::`):

```
class A {
    public static function who() {
        echo __CLASS__;
    }
    public static function test() {
        self::who();
    }
}
class B extends A {
    public static function who() {
        echo __CLASS__;
    }
}
B::test(); // prints A, not B!
```

- nested classes (a class defined within a class) are not supported

```
class a {
    function nextFoo() {
        class b {} //not supported
    }
}
```

- globals aren't always "globals":

```
$var1 = "Example variable";
$var2 = "";
function global_references($use_globals)
{
    global $var1, $var2;
    if (!$use_globals) {
        $var2 =& $var1; // visible only inside the function
    } else {
        $GLOBALS["var2"] =& $var1; // visible also in global context
    }
}
global_references(false);
echo "var2 is set to '$var2'\n"; // var2 is set to ''
global_references(true);
echo "var2 is set to '$var2'\n"; // var2 is set to 'Example variable'
```

- there's no module / package concept: only include files, "ala C"
 - Much PHP functionality is provided via compiled modules written in C.
- no way to store 64-bit integers in a native data type on a 32-bit machine, leading to inconsistencies (intval('9999999999') returns 9999999999 on a 64-bit machine, but returns 2147483647 on a 32-bit machine)
- The class that represents a file is called: `SplFileObject`.
- `SplFileObject` both extends **and** is a property of the file meta object `SplFileInfo`. Can't choose between inheritance and composition? LETS USE *BOTH*!?
- PHP is just about the only *programming language* out there with a *configuration file*. Things like `short_open_tags`, which would only make sense to be set per-application, are set *by the administrator*, for all the apps he installs!
- this makes my head explode:

```
in_array("foobar", array(0)) === true
```

- This is because when doing non-strict comparisons, the string "foobar" is coerced into an integer to be compared with the 0. You need to use the strict flag for `in_array` to work as expected, which uses `===` instead of `==`, apparently. You know, for true equality, not just kind of equality.
- `php.ini` can change the whole behavior of your scripts thus not making them portable between different machines with different settings file. Also, this is the only scripting language with a settings file.
- `null`, `""`, `0`, and `0.0` are all equal.
- Numbers beginning with `0` are octals, so `08`, `09`, `012345678` and the like should raise an error. Instead, it just ignores any digits after the first 8 or 9: `08 == 0`, `08 != 8`, `0777 == 07778123456` (fixed in PHP 7).
- there's no integer division, just floating point one, even if both operands are integers; you must truncate the result to have back an integer (there is `intval()` function as of PHP 7)

Fixed in currently supported versions

Those points were valid before PHP 5.5. Older versions are not supported (as of 04/19/2016):

- Fatal errors do not include a traceback or stack trace (fixed in PHP 5)
- Using `[]` or `{}` to a variable whose type is not an array or string gives back `NULL` (fixed in PHP 5)
- There are constructors but not destructors (fixed in PHP 5)
- There are "class methods" but not class (static) variables (fixed in PHP 5)
- References in constructors doesn't work (fixed in PHP 5):

```
class Foo {
    function Foo($name) {
        // create a reference inside the global array $globalref
    }
}
```

```

    global $globalref;
    $globalref[] = &$this;
    $this->setName($name);
}
function echoName() {
    echo "\n", $this->name;
}
function setName($name) {
    $this->name = $name;
}
}
$bar1 = new Foo('set in constructor');
$bar1->setName('set from outside');
$bar1->echoName(); // prints "set from outside"
$globalref[0]->echoName(); // prints "set in constructor"

// It's necessary to reference the value returned from new to have back a reference to the same obj,
$bar2 =& new Foo('set in constructor');
$bar2->setName('set from outside');
$bar2->echoName(); // prints "set from outside"
$globalref[1]->echoName(); // prints "set from outside"

```

- A method defined in a base can "magically" become a constructor for another class if its name matches the class one (fixed in PHP 5):

```

class A
{
    function A()
    {
        echo "Constructor of A\n";
    }
    function B()
    {
        echo "Regular function for class A, but constructor for B";
    }
}
class B extends A
{
}
$b = new B; // call B() as a constructor

```

- Exceptions in the `__autoload` function cannot be caught, leading to a fatal error (fixed in PHP 5.3)
- No support for closures; `create_function` does not count since it takes a string as the argument and doesn't reference the scope it was created in (fixed in PHP 5.3)
- If a function returns an Array, you just can not write (fixed in PHP 5.4):

```

$first_element = function_returns_array()[0]; //Syntax ERROR!!
$first_element = ( function_returns_array() )[0]; //This neither!!
//Instead you must write
$a = function_returns_array();
$first_element = $a[0];

```

Perl 5 sucks because

- *Perl is worse than Python because people wanted it worse.* Larry Wall, 14 Oct 1998

- use strict -- really, it should be 'use unstrict' or 'use slop' (as in billiards/pool) that changes things, and that itself should never ever be used. Ever. By anyone. 'strict' mode should be the default.

```
use strict;  
use warnings;
```

- Sigil variance is hugely annoying

```
my @list = ("a", "b", "c");  
print $list[0];
```

- no parameter lists (unless you use Perl6::Subs)

```
sub foo {  
    my ($a, $b) = @_;  
}
```

- dot-notation for methods, properties, etc is a good thing, especially when so many other C-styled languages do it and Perl is one of the random ones that doesn't.
- The type system should be revamped. There is only support for (Scalar, Array, Hash, etc) and not support for (Int, Str, Bool, etc).
- It is too hard to determine the type of a Scalar, e.g there is no easy way to determine if a variable is a string.
- Just try explaining a ref to a C programmer. They will say "oh, it's a pointer!" except it isn't. Not really. It's *like* a pointer, or so I hear. As a Perl programmer, I know that it's not quite a pointer, but I don't know exactly what a pointer is (while I do know what a ref is, but I can't explain it -- neither could Larry Wall: He called it a 'thingy').
- The regular expression syntax is horrid.
- there's rarely a good place to put the semicolon after a here-doc.

```
my $doc = <<"HERE";  
    But why would you?  
HERE  
print $doc;
```

- it generally takes you ten years to figure out how to call functions or methods inside double-quoted strings like any other variable. If you're reading this, though, you get a break: "You do it @{{sub{'like'}}} this. Easy"
- just like Ruby it has redundant keyword "unless". You can do "if not" at least in three ways, which can lead to confusion and spoiling readability of code:
 1. if(!expression)
 2. if(not expression)
 3. unless(expression)
- i cant say if(\$a==\$b) \$c=\$d ; instead i have to say either:

1. `$c=$d if($a==$b) ; or`
 2. `if($a==$b) { $c=$d; }`
- what's with all the \$,@,%,& things in front of variables? It takes a lot of effort to type those redundancies every single time ... C and most others let you define a type just once, and then you can use it without reminding yourself what it is. In any case, that Perl lets you change the thing depending on what you want to do so it's even more stupid to use say @ and \$ in front of the same var.
 - You will not understand your program when you revisit it 6 months later.

Python sucks because

- mandatory self argument in methods, although that means you can use methods and functions interchangeably.
 - even so, there are cases where `a.b()` would not necessarily call `b` as a method of `a`; in other words, there are cases when `a.b()` will not send `a` as `b`'s "self" argument; like:

```
class NoExplicit:
    def __init__(self):
        self.selfless = lambda: "nocomplain"

    def withself(): return "croak" #will throw if calld on an _instance_ of NoExplicit

a = NoExplicit ()

print(a.selfless()) #won't complain
print(a.withself()) #will croak
```

That is, even though an expression of the form `a.b()` looks like a method call, it is not always so ; arguably this is against the policy of "everything must be explicit and predictable as much as possible" python so vehemently tries adhering to .

- Since tuples are represented with parentheses for clarity and disambiguation, it's a common misunderstanding that the parentheses are needed, and part of the syntax of tuples. This leads to confusion, as tuples are conceptually similar to lists and sets, but the syntax is different. It's easy to think it's the parentheses that makes it a tuple, when it is in fact the comma. And if that wasn't already confusing enough, the empty tuple has no comma, only parentheses!

```
(1) == 1           # (1) is just 1
[1] != 1           # [1] is a list
1, == (1,)         # 1, is a tuple
(1) + (2) != (1,2) # (1) + (2) is 1 + 2 = 3
[1] + [2] == [1,2] # [1] + [2] is a two-element list
isinstance((), tuple) == True # () is the empty tuple
```

- No labeled break or continue (<http://www.python.org/dev/peps/pep-3136/>)
- the body of lambda functions can only be an expression, no statements; this means you can't do assignments inside a lambda, which makes them

pretty useless

- no switch statement -- you have to use a bunch of ugly if/elif/elif ... tests, or ugly dispatch dictionaries (which have inferior power).
- the syntax for conditional-as-expression is awkward in Python (x if cond else y). Compare to C-like languages: (cond ? x : y)
- no tail-call optimization, ruining the efficiency of some functional programming patterns
- no constants
- There are no interfaces, although abstract base classes are a step in this direction
- There are at least 5 different types of (incompatible) lists [1] (<http://jacobian.org/writing/hate-python/>)
- Inconsistency between using methods/functions - some functionalities require methods (e.g. list.index()) and others require functions (e.g. len(list))
- No syntax for multi-line comments, idiomatic python abuses multi-line string syntax instead

```
"""
```

- Function names with double underscore prefix and double underscore suffix are really really ugly

```
__main__
```

- String character types in front of the string are really ugly

```
f.write(u'blah blah blah\n')
```

- Python 3 allows you to annotate functions and their arguments, but they don't do anything by default. Since there is no standard meaning for these, usage varies by toolkit or library, and you can't use it for two different things at once. Python 3.5 tried to fix this by adding optional type hints to the standard library, but since the majority of python is unannotated and you can only annotate functions, it's mostly useless.
- Generators are defined by using "yield" in a function body. If python sees a single yield in your function, it turns into a generator instead, and any statement that returns something becomes a syntax error.
- Python 3.5 introduces keywords async and await for defining coroutines. Python pretended to have coroutines by abusing generators with yield and yield from, but instead of fixing generators, they added another thing. Now there are asynchronous functions, as well as normal functions and generators, and the rules for what keywords you can use inside of them are even more complex[2] (<https://www.python.org/dev/peps/pep-0492/#list-of-functions-and-methods>). Unlike generators, where anything containing a yield becomes a generator, anything that uses coroutines has to be prefixed with async, including def, with and for.
- The super() call is a special case of the compiler that works differently if renamed [3] (<https://stackoverflow.com/a/19609168/539465>)
- The standard library uses inconsistent naming schemes e.g.:

`os.path.expanduser` and `os.path.supports_unicode_filenames` (former does not separate words by underscore, while the latter does)

Fixed in Python 3

- `!=` can also be written `<>` (see php).
- incomplete native support for complex numbers: both `(-1)**(0.5)` and `pow(-1, 0.5)` raise an error instead of returning `0+1j`.
- the ability to mix spaces and tabs raises confusion around whether one white space or one tab is a bigger indent.

Ruby sucks because

- `String#downcase`? Who calls it "downcase?" It's called "lower case," and the method should be called "lowercase" or "lower". And `String#upcase` should have been called "uppercase" or "upper". It does not actually make Ruby suck, it's just a matter of personal taste. Still a better story than tw... PHP.
- Unicode support should have been built in from 1.0, not added after much complaining in 1.9/2.0 in 2007
- No support for negative / positive look-behind in regular expressions in 1.8
- Regular expressions are always in multi-line mode
- (No longer true since Ruby 2.0!) No real support for arbitrary keyword arguments (key=value pairs in function definitions are positional arguments with default values).
- Using `@` and `@@` to access instance and class members can be unclear at a glance.
- There are no smart and carefully planned changes that can't break compatibility; even minor releases can break compatibility: See "Compatibility issues" and "fileutils" (http://svn.ruby-lang.org/repos/ruby/tags/v1_8_7/NEWS). This leads to multiple recommended stable versions: both 1.8.7 and 1.9.1 for Windows (<http://www.ruby-lang.org/en/downloads/>). Which one to use?
- Experimental and (known to be) buggy features are added to the production and "stable" releases: See "passing a block to a Proc" (http://svn.ruby-lang.org/repos/ruby/tags/v1_8_7/NEWS).
- The documentation is unchecked: it has dead links, like Things Any Newcomer Should Know (<http://www.rubygarden.org/ruby?ThingsNewcomersShouldKnow>)
- There's some minor gotchas. `nil.to_i` turns `nil` into 0, but 0 does not evaluate as `nil`. `nil.to_i.nil? #=> false`
- `String#to_i` just ignores trailing characters, meaning: `"x".to_i == 0`
- Ruby allows users to modify the built in classes, which can be useful, but limited namespace means addons can conflict. Experienced programmers know to add functionality through modules rather than monkey patching the built in classes, but is still prone to abuse. This has been resolved in ruby 2.0
- Aliased methods in the standard library make reading code written by others more confusing, if one is not yet well familiar with basic stuff. E.g.

Array#size/Array#length, Array#[]/Array#slice

- Mutable types like arrays are still hashable. This can cause a hash to contain the same key twice, and return a random value (the first?) when accessing the key.
- Omitting parenthesis in function calls enable you to implement/simulate property setter, but can lead to ambiguities.
- Minor ambiguities between the hash syntax and blocks (closures), when using curly braces for both.
- Suffix-conditions after whole blocks of code, e.g. `begin ... rescue ... end if expr`. You are guaranteed to miss the `if expr` if there are a lot of lines in the code block.
- The `unless` keyword (acts like `if not`) tends to make the code harder to comprehend instead of easier, for some people.
- Difference between unqualified method calls and access of local variables is not obvious. This is especially bad in a language that does not require you to declare local variables and where you can access them without error before you first assign them.
- "Value-leaking" functions. The value of the last expression in an function is implicitly the return value. You need to explicitly write `nil` if you want to make your function "void" (a procedure). Like if you really care about that return value.
- pre-1.9: No way to get `stdout`, `stderr` and the exit code (all of them at once) of a sub process.
- `` `` syntax with string interpolation for running sub processes. This makes shell injection attacks easy.
- Regular expressions magically assign variables: `$1`, `$2`, ...
- Standard containers (`Array`, `Hash`) have a very big interfaces that make them hard to emulate. So don't emulate - inherit instead. `class ThingLikeArray < Array; end`
- Symbols and strings are both allowed and often used as keys in hashes, but `"foo" != :foo`, which led to inventions like `HashWithIndifferentAccess`.
- Parser errors could be more clear. "syntax error, unexpected `kEND`, expecting `$end`" actually means "syntax error, unexpected keyword 'end', expecting end of input"
- Symbols are unequal if their encoded bytes are unequal even if they represent the same strings.

Flex/ActionScript sucks because

- The `String` class is defined as `final`, so if you want to add a function to the woefully incomplete `String` class, like `startsWith` or `hashCode`, you have to implement pass thrus for all the `String` methods.
- Method variables are scoped to the whole method, not to the current code block. (Note that this also sucks in JavaScript)
- No `abstract` keyword, and no support for protected constructors, making compile time abstraction checks impossible.
- Publically visible inner classes are not supported, though private inner

classes can be hacked in.

Scripting languages suck because

- They are too domain specific, so when you want to go out of your domain you have to use some other scripting language, and when you want to extend the language itself or doing anything really advanced you have to use a systems language. This leads to a big mess of different programming languages.

C sucks because

- Manual memory management can get tiring.
- String handling is manual memory management. See above.
- The support for concurrent programming is anemic.
- Terribly named standard functions: `isalnum`, `fprintf`, `fscanf` etc.
- The preprocessor.
- Not feeling your smartest today? Have a segfault.
- Lack of good information when segfault occurs... the "standard" GNU toolchain doesn't tell you the causes of segfaults in the standard runtime.
- If that code is legacy, you're in for two days of not looking like a hero.
- The standard library only covers basic string manipulation, file I/O, and memory allocation. And `qsort()` for some reason. You get nothing else that's truly portable.
 - Rolling your own containers (likely inferior to a standard, highly-optimised version) is something you better get used to doing for every new application, or using overly complicated existing frameworks that take over your application (`nspr`, `apr`, `glib`...).
 - There is also no standard for network programming. Most platforms use the Berkeley sockets API, but it varies wildly between vendors. Microsoft in particular has many striking differences in their API. Good luck finding a portable $O(1)$ asynchronous socket selection API though; there's none (`select()` and `poll()` have $O(n)$ behaviour).
- Variables are not automatically initialised to zero, unless they're static, because "it's more efficient."
- Want to use C99 portably? Sure, but if you need to use MSVC older than 2015 (very likely at some dev houses), you're screwed.
- Want to use C11 portably? Sure, but pretty much nothing besides GCC and clang support it. Sorry MSVC users.
- Variable length arrays are usually placed on the stack, meaning that passing in too large of a size is unsafe; it will just stack overflow the moment you use it, made worse by the fact **there is no way to tell your allocation is too big**.
- `malloc()` takes one size parameter (size in bytes), but `calloc()` takes two (number of elements and element size in bytes).
- Undefined behaviour is the name of the game in C. By undefined they mean "welcome to do whatever the vendor wants, and not necessarily what you want." The worst part is **it's impossible to reliably find all undefined behaviour in a large program** (<http://blog.llvm.org>

/2011/05/what-every-c-programmer-should-know_14.html).

- `int a = 0; f(a, ++a, ++a);` is not guaranteed to be `f(0, 1, 2)`. The order of evaluation is undefined.
- Another surprise that bites people: `i[++a] = j[a];`, since which side is evaluated first is undefined.
- Signed overflow is technically undefined behaviour. It just happens to work on most platforms.
- Shifting by more than N bits on an `intN_t` type is undefined behaviour.
- Casting an `int *` to a `float *` then dereferencing it is undefined behaviour. You have to use `memcpy()`.
- Dereferencing a NULL pointer is undefined behaviour. It doesn't have to trap - you can actually get valid memory out of it!
- Casting between function pointers and data pointers is technically undefined behaviour; on some platforms, they're different sizes (why a programmer needs to care in C is left as an exercise for the reader).
- Casting `void (*)()` to `int (*)(int, const char *)` should be undefined right? Nope! All function pointers can be cast to one another. Actually calling the function after the cast if it is not the correct type is undefined behaviour, though (as you'd rightfully expect).
- Casting something like, say, `FILE *` to `double *` and back again is **not** undefined, so long as you don't dereference the pointer in-between. All data pointers are equivalent.
- Debugging optimised code can sometimes make no sense due to aggressive compiler optimisation.
- Data safety in multi-threaded programs is not really guaranteed by the language; things you think are atomic aren't necessarily without using (C11, but see above) atomics. Getting torn reads and writes is extremely easy, and no tools will ever warn you it's happening.
- void pointers are a drag; there's no way to determine what is actually behind them without some sort of user-defined tagging scheme that by definition cannot comprise all types. The solution is apparently "be careful."
- The type system in general is useless since you can cast (almost) anything to anything else.

C++ sucks because

Some points in this section have been disputed on the discussion page.

- It is backward compatible with C. See above.
- But still with subtle differences that make some C code unable to compile in a C++ compiler.
- The standard libraries offer very poor functionalities compared to other languages' runtimes and frameworks.
- C++ doesn't enforce a single paradigm. Neither procedural nor object-oriented paradigms are enforced, resulting in unnecessary complication. [Some people consider this as an advantage.]
- Too hard to implement and to learn: the specification has grown to over

1000 pages.

- Everybody thus uses a different subset of the language, making it harder to understand others' code.
- Not suitable for low level system development (<https://web.archive.org/web/20140111204548/http://msdn.microsoft.com/en-us/windows/hardware/gg487420.aspx#EFE>) and quickly becomes a mess for user level applications.
- The standard does not define exception handling and name mangling. This makes cross-compiler object code incompatible.
- No widely used OS supports the C++ ABI for syscalls.
- What is 's', a function or a variable?

```
std::string s();
```

- Answer: it's actually a function; for a variable, you have to omit the parentheses; but this is confusing since you use usually the parentheses to pass arguments to the constructor.
- The value-initialized variable 's' would have to be:

```
std::string s; /* or */ std::string s{};
```

- There is *try* but no *finally*.
 - Considered a "feature" because RAII is supposed to be the main means of resource disposal in C++. Except a lot of libraries don't use RAII.
- Horrid Unicode support.
- Operators can be overloaded only if there's at least one class parameter.
 - This also makes impossible concatenating character array strings, sometimes leading programmers to use horrible C functions such as `strcat`.
- `catch (...)` doesn't let you know the type of exception.
- `throw` in function signatures is perfectly useless.
- The exception system is not integrated with the platform: dereferencing a `NULL` pointer will not throw a C++ exception. [Some people consider this as an advantage.]
- `mutable` is hard to use reasonably and, since it fucks up `const` and thus thread safety, it can easily bring to subtle concurrency bugs.
- Closures have to be expressed explicitly in lambda expressions (never heard about anything like that in any functional language).
 - You can use `[=]` and enclose everything, but that still adds verbosity.
- The nature of C++ has led developers to write compiler dependent code, creating incompatibility between different compilers and even different versions of the same compiler.
- An `std::string::c_str()` call is required to convert an `std::string` to a `char*`. From the most powerful language ever we would have all appreciated a overloaded operator `const char* () const`.
- Developers may have to worry about optimization matters such as whether declaring a function `inline` or not; and, once they've decided to do it, it is only a suggestion: the compiler may decide that was an incorrect suggestion and not follow it. What's the point? Shouldn't developers worry

about optimization matters?

- To rectify this nonsense many compilers implement `__forceinline` or similar.
- Templates are Turing-complete, hence compilers have to solve the halting problem (undecidable) to figure out whether a program will even compile.
- Unused global symbols do not generate any warnings or errors, they are compiled and simply increase the size of the generated object file.
- Changes that could actually standardise key functionality or substantially improve language usability often (<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4478.html>) take (<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4377.pdf>) a (<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4514.pdf>) TS (<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/n4592.pdf>) detour (<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0267r2.pdf>) to (<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3820.html>) nowhere (<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/n4569.pdf>). Technical Specifications seem to be where good ideas go to die, often for political reasons, or some country didn't like it.
- The fact `export` was never supported (<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1426.pdf>) means you have to include the entirety of a template in your library header. Traditional C (and most other C++ code) separates implementation and prototypes.
- Every problem with the language is resolved by adding a new feature.

.NET sucks because

- `SortedList` uses key-value pairs. There is no standard .NET collection for a list that keeps its items in sort order.
- Changing a collection invalidates *all* iteration operations in *every* thread, even if the current item is unaffected. This means every `foreach` can and will throw an exception when you least expect it. Unless you use locks which in turn can cause deadlocks. Can be circumvented by using official concurrent or immutable collections or by iterating by using `"for"` instead of `"foreach"`.
- The MSDN documentation of the (hypothetical) `GetFrobnicationInterval` would explain that it returns the interval at which the object frobnicates. It will also state that the method will throw an `InvalidOperationException` if the frobnication interval cannot be retrieved. You will also find two "community comments", one of which will contain line noise and the other will inquire about what frobnication is at room temperature in broken English.

C# sucks because

- ECMA and ISO standards of C# have been updated since C# 2.0, since then only Microsoft's specification exists.
- `i++.ToString` works, but `++i.ToString` does not.
- Parameters are not supported for most properties, only indexers, even though this is possible in Visual Basic .NET.

- The conventions are difficult to code review and deviate from other popular language conventions of similar style:
 - Almost everything is in pascal case (SomeClass, SomeConstantVariable, SomeProperty)
 - Unable to differentiate between 'extends' and 'implements' without using Hungarian-like notation such as IInterface
- Promotes slop (variant types and LINQ, while powerful adds a lot of clutter)
- "out" parameters, with syntax sugar.
- You can't override a virtual or abstract property that has a getter OR a setter with one that has a getter AND a setter, making life quite difficult in the realm of property inheritance.
 - This does not apply to interfaces though, read <http://stackoverflow.com/questions/82437/why-is-it-impossible-to-override-a-getter-only-property-and-add-a-setter>.
- You can't perform any operations, even simple arithmetic, with objects inside a generic method (e.g. `T plus<T>(T t1, T t2) { return t1+t2; }`)
- You can't assign a new value inside a foreach loop (e.g. `foreach(int i in vec) { i = i+1; }`).
- Implementing IDisposable properly (<https://msdn.microsoft.com/en-us/library/fs2xkftw.aspx>) is too complicated. In particular, a naive implementation will not dispose if the finalizer is called by the garbage collector.
- Inconsistent numeric types: `short` is signed and `ushort` is unsigned; `int` is signed and `uint` is unsigned; `long` is signed and `ulong` is unsigned — but `byte` is *unsigned* and `sbyte` is signed.

VB.NET sucks because

- Option Strict Off: It enables implicit conversion wherever the compiler thinks it is appropriate. e.g. `Dim a As Integer = TextBox1.Text` ' (String is converted to Integer by using `Microsoft.VisualBasic.CompilerServices.Conversions.ToInteger`)
- Option Explicit Off: Automatically declares local variables of type Object, wherever an undeclared variable is used. Commonly used with Option Strict Off.
- On Error Goto and On Error Resume Next: These are procedural ways to hide or react to errors. To actually catch an Exception, one should use a Try-Catch-Block.
- Lots of functions for downwards-compatibility, like `UBound()`, `MkDir()`, `Mid()`, ... These can be hidden by removing the default-import for the `Microsoft.VisualBasic-Namespace` in the project-settings.
- The My-Namespace (except for `My.Resources` and `My.Settings`, which can be very useful). Everything in this Namespace is a less flexible version of an existing Member. e.g. `My.Computer.FileSystem.WriteAllText` vs `System.IO.File.WriteAllText`.
- Modules, because their members clog up intellisense because Module-Members are visible as soon as the Module is visible.
 - Extension-Methods can only be defined in Modules.
 - Extension-Methods cannot be applied to anything typed Object, even

when late-binding (Option Strict Off) is disabled. See this [StackOverflow article \(http://stackoverflow.com/questions/3227888/vb-net-impossible-to-use-extension-method-on-system-object-instance\)](http://stackoverflow.com/questions/3227888/vb-net-impossible-to-use-extension-method-on-system-object-instance).

- The Microsoft.VisualBasic.HideModuleNameAttribute. It is unnecessary because of the nature of Modules.
- Default-Instances for Forms. It is valid to write

```
Form2.InstanceMethod()
```

Instead of

```
Dim Form2Instance As New Form2
Form2Instance.InstanceMethod()
```

because what actually is compiled is:

```
MyProject.Forms.Form2.InstanceMethod()
```

- String-Concatenation is possible with + and &, instead of just &. + confuses new programmers, especially when they have Option Strict set to Off.

VBA sucks because

- Array indexes start at zero, but Collection indexes start at one.
- Classes don't have constructors that can accept arguments.
- You can't overload methods.
- No inheritance.
- Supports GoTo
- `OnError Resume Next` - Yeah... It does exactly what it sounds like. Encountered an error? No problem! Just keep chugging along at the next line.
- Default Properties

The following two lines do not do the same thing.

```
Dim myRange As Variant
myRange = Range("A1")
Set myRange = Range("A1")
```

One sets `myRange` equal to the value of "A1", the other sets it to the actual Range object.

Objective-C sucks because

- No real use outside of programming for OS X and iOS that can't be done by another C-derived language, meaning your skill set doesn't extend past a certain market.
 - Using Objective-C portably is an oxymoron; GNUStep is a window manager for the fringe on Linux, and there's nothing really there for Windows at all... about the only thing that works is the compiler.
- No operator overloading.
- A "work-around" exists for method overloading.
- Tries to shoehorn the dynamically-typed Smalltalk into the statically-typed C.
- No stack-based objects.
- Syntax is very strange compared to other languages (I have to put a @ before the quotes when making a string?! Methods are called like this unless you have a single argument?!? [methodName args];)
- <http://fuckingblocksyntax.com>
- There is no specification at all. No one (except maybe some LLVM developers and Apple) really knows what's going on under the hood.
- Could randomly crash if you return SEL from method [4] (<http://stackoverflow.com/a/20058585/1437441>)
- Awful type system. Types are more recommendations than types.
- Objective-C++ and the unholy horrors that arise from that
 - Objective-C and C++ classes can't inherit from one another.
 - C++ namespaces cannot interact with Objective-C code at all.
 - C++ pass by value can't be applied to Objective-C objects in C++ functions; you have to use pointers.
 - C++ lambdas and Objective-C blocks are different and not interchangeable.
 - Objective-C classes can't have members that are C++ classes that lack a default constructor, or that have one or more virtual methods... except via pointers and allocating via new.
- Has no namespaces and instead encourages to use prefixes (two characters mostly) every class' name to prevent naming collision
- Objective-C has no standard library. Apple controls the de-facto one, and everything's prefixed with NS*.
- Apple's focus on Swift (which is itself yet another platform-specific language) means Objective-C is being left to rot on the vine.

Java sucks because

Syntax

- Overly verbose.
- Java has unused keywords, such as `goto` and `const`.
- There's no operator overloading... except for strings. So for pseudo-numeric classes, like `BigInteger`, you have to do things like `a.add(b.multiply(c))`, which is really ugly.
- There are no delegates; everytime you need a function pointer you have to implement a factory design.
- Arrays don't work with generics: you can't create an array of a variable

type new T[42], boxing of array is required to do so:

```
class GenSet<E> { Object[] a; E get(int i){return a[i];}}
```

- No properties. Simple class definitions are 7 to 10 times as long as necessary.
- No array or map literals. Array and Map are collection interfaces.
- No var-keyword for inferring local types (like in C#). Please mind that it is antidesign example and class names should never be that long. Example:

```
// In Java
ClassWithReallyLongNameAndTypeParameter<NamingContextExtPackage> foo = new ClassWithReallyLongNameAnd
// In C# | Could easily be just:
var foo = new ClassWithReallyLongNameAndTypeParameter<NamingContextExtPackage>();
// In Java | Same goes for function calls:
SomeTypeIHaveToLookUpFirstButIActuallyDontReallyCareAboutIt result = getTransactionResult();
```

- It's, like, impossible to write Java without an IDE with autocompletion, code generation, import management and refactoring.
- No pairs or triples. Returning two values from a function or placing a pair in a set warrants a new class in a new file. Generic Pair class leads to hairy types all over the place.

Fixed in Java 7 (2011)

- ~~The catch clauses can contain only one exception, causing a programmer to rewrite the same code N times if she wants to react the same way to N different exceptions.~~
- ~~There is no automatic resource cleanup; instead we get have five lines of "allocate; try {...} finally { cleanup; }"~~

Model

- No first-class functions and classes.
- Checked exceptions are an experiment that failed (<http://www.artima.com/intv/handcuffs.html>).
- There's int types and Integer objects, float types and Float objects. So you can have efficient data types, or object-oriented data types.
 - Number base-class doesn't define arithmetic operations, so it is impossible to write useful generics for Number subclasses.
- Before Java 8, using only interfaces to implement multiple inheritance didn't allow sharing common code through them.

Library

- Functions in the standard library do not use consistent naming, acronym and capitalization conventions, making it hard to remember exactly what the items are called:
 - This is actually part of the standard, you write all of them in uppercase if it has 3 or less letters, and only the first if it has more so, RpgGame and not RpgGame, and TLSConnection but you should use Starttls.
- The design behind Cloneable and clone is just broken (<http://www.artima.com/intv/bloch13.html>).

- Arrays are objects but don't properly implement `.toString()` (if you try to print an array, it just prints hash code gibberish) or `.equals()` (arrays of the same contents don't compare equal, which causes a headache if you try to put arrays into collections)
- Before Java 8, useful methods like sorting, binary search, etc. were not part of Collection classes, but part of separate "utility classes" like `Collections` and `Arrays`
- Why is `Stack` a class, but `Queue` an interface?
 - `Stack` belongs to old collection API and should not be used anymore. Use `Deque`(interface) and `ArrayDeque`(implementation) instead.
- Code is cluttered with type conversions. Arrays to lists, lists to arrays, `java.util.Date` to `java.sql.Date`, etc.
- The `Date` API is considered deprecated, but still used everywhere. The replacement `Calendar` is not.
- Before Java 8 there was no string join function.
- The reflection API requires multiple lines of code for the simplest operations.
- `(a|[^d])` regex throws `StackOverflowException` on long strings
- No unsigned numeric types

Disputed

The points in this section have been disputed on the discussion page.

- Nearly everything is wrapped with objects and many things must be buffered, even those things that don't really need to be objects or be buffered. (*examples?*)
- Some interfaces such as `Serializable` and `RandomAccess` are used just like annotations: they are empty, and when implemented their only purpose is to indicate some semantics.
- Initialization blocks (both static and non-static) can't throw checked exceptions.
- Arrays are not type safe: `Object[] foo = new String[1]; foo[0] = new Integer(42);` compiles fine but fails at runtime
- Unicode escapes can have unexpected effects, because they are substituted *before* the code is parsed, so they can break your code, for example: (1) if a line-end comment contains a `\u000A` (line return), the rest of the comment won't be on the same line, and thus won't be in the comment anymore; (2) if a string literal contains a `\u0022` (double quote), it ends the string literal, and the rest of the string is now in the actual code; (3) if a `\u` appears in a comment, and it is not a valid escape (e.g. `"c:\unix\home"`), it will cause a parsing error, even though it is in a comment
- Convenience functions must be overloaded for every fundamental type (e.g. `max(float,float)`, `max(double,double)`, `max(long,long)`)

Backbase sucks because

- Oh really, this subject could take a whole new wiki by itself.

XML sucks because

- Attributes are generally limited to unstructured data.
- Its too verbose, too hard for programs to parse, and too hard for people to read.
- Boilerplate verbosity does not play nice with subversion systems like Git. Entries of the same type start and end the same way even if their content is entirely different. This confuses auto-merge programs (which often don't understand how tags work) into mistaking one entry for another and often requires merging to be done manually to avoid corruption.
- It confuses metadata and content.
- It is used for data interchange when it is really just a data encoding format.
- No default way to encode binary.

XSLT/XPath sucks because

- It starts numbering from 1. Unlike *every single other* major programming language in use today. Unlike the XML DOM.
- XPath has date manipulation functions to get the second, minute, hour, date, month, and year from a date-time. But it has no function to get the day of the week, so it's completely useless.
- There is no way to modularize or abstract any XPath expressions, resulting in lots of copied and pasted code.
 - conditionals in `test=""` attributes of `<xsl:if>` and `<xsl:when>` items.
 - sorting conditions in `<xsl:sort>`
- When your context is the contents of a node-set, the `key()` functions defined on the entire input XML do not work. What's dumber, no error message is produced; your `select="key(...)"` just silently returns the empty set. It should at least say "key() does not work inside node-sets" or perhaps "no such key in context node-set"
- The `select=""`, `value=""` and `match=""` attributes do basically the same thing. And their use is arbitrarily exclusive; you must use the correct attribute in the correct place, and if you use the wrong one, the node fails without any warning. These three attributes should have the same name.
- If you import a function (like `str:replace()`) but fail to import it correctly or fully (like leaving off the namespace), and then call that function, no error is raised whatsoever. The function simply evaluates to its last argument. How could this **ever** be desirable behavior? If I'm calling a function that's somehow not available, obviously that is **always** a programmer error, and some warning should be raised.
- There's no way to construct a custom set of values and then iterate over it at runtime, although there is a way to construct a single custom value and then operate on it at runtime. In other words, the language has no list/array/tuple/dict/hash/set/iterable/collection type.
- It allows '-' in identifiers, but the parser isn't smart enough to figure out when you mean 'minus' instead of - . If you're going to allow '-' as an identifier character and as an operator, at least make it so that the string following the identifier character '-' has to follow the standard identifier

pattern, `[a-zA-Z_][a-zA-Z0-9_]*`. Don't make **this one use of whitespace** significant in a language where whitespace is usually insignificant around operators. **Nobody** is ever going to want a variable name like `$foo-100`, **because it looks just like `$foo - 100`**.

- `$foo-bar` is rightly interpreted as a variable name
- `$foo - 100` is rightly interpreted as subtraction
- `$foo+100` and `$foo + 100` are rightly interpreted as addition
- `$foo-100` is wrongly interpreted as a variable name
- There is no concept of types whatsoever. Everything is fundamentally a string. This means that even things that are intrinsically typed are treated as strings fundamentally. For example, sorting on the number of child nodes sorts by string order, not numeric order, even though counting is an intrinsically numeric operation.

```
<xsl:sort select="count(*)"/>
<!-- sorts like this: 10, 11, 1, 23, 20, 2, 37, 33, 31, 3, 4, 5, 6, 78, 7, 9 -->
```

- There are too many levels of syntactic and semantic interpretation:
 1. Parse XML syntax (ensure that all nodes are closed, etc)
 2. Parse XSL syntax (ensure that nodes that must be under/contain other nodes are present, verify that all `xsl:foo` node names are valid, etc)
 3. Parse XSL semantics (find the correct attributes under each node type, etc)
 4. Parse XPath syntax (entirely contained inside attribute values, cannot be parsed earlier)
 5. Parse XPath semantics

CSS sucks because

- Why is there `hsla()` but no `hsva()`?
- `text-align: justify`; really means "left justify". No way to right justify or center justify.
- `vertical-align: middle`; doesn't work with block elements, although it works with inline elements and table elements. This leads to people suggesting `display: table`; and `display: table-cell`; and it means you have to style the wrapper as well. WTF.
- Horizontally aligning block elements was not intended to be supported and is only hackish at best (`margin: 0 auto`).
- Can float an item to the left or the right. But you cannot float items to the center.
- `float`: is only horizontal; there is no equivalent vertical operation. (OK, it's actually a flow operation, go figure.)
- Similarly, there is no vertical equivalent to `clear`.
- No way to modularize or programmatically generate colors. If you want text and borders to use the same color, you have to write that color twice.
- No way to modularize or programmatically generate lengths. CSS3 introduces `calc` (<http://hacks.mozilla.org/2010/06/css3-calc/>) in the CSS Values and Units Module (<http://www.w3.org/TR/css3-values/>), but you still

can't say something like `{ width:50% - 2px; }`

- The CSS spec is contradictory with regard to identifiers:
 - The syntax says that identifiers don't allow uppercase characters anywhere but in the first character:
 - `ident {nmstart}{nmchar}*`
 - `nmstart [a-zA-Z]|{nonascii}|{escape}`
 - `nmchar [a-z0-9-]|{nonascii}|{escape}`
 - Section "4.1.3 Characters and case:" says:
 - In CSS2, identifiers (including element names, classes, and IDs in selectors) can contain only the characters `[A-Za-z0-9]` and ISO 10646 characters 161 and higher, plus the hyphen (-); they cannot start with a hyphen or a digit."
- **Why** diverge from the standard identifier format, `[a-zA-Z_][a-zA-Z0-9_]*`, which has been in use since the 1970s?
- Will we ever have alpha masks? Webkit does it but...
- Supporting vendor prefixes suck. Pretending -webkit- is the only vendor prefix sucks even more.
- There's SASS, LESS and Stylus. Take every single feature. Every one is a CSS wtf. (indent-based syntax should be optional, though).
- No parent elements selectors, even in CSS3. Maybe we finally can accomplish this in CSS4 (<http://www.w3.org/TR/selectors4/#subject>).

See also CSS Considered Unstylish (http://www.cybergrain.com/archives/2004/12/css_considered.html) and Incomplete List of Mistakes in the Design of CSS (<https://wiki.csswg.org/ideas/mistakes>).

Fixed in CSS3

- Hello? Rounded corners? Mac OS had them in 1984. (CSS3 introduces `border-radius:`)
- Can only set one background image. (CSS3 supports multiple backgrounds)
- Can't specify stretchable background images. (CSS3 introduces `background-size:` and `background-clip:`)
- No way to make vertical or angled text. (CSS3 introduces `rotate`)

Scala sucks because

- `scala.collection` hierarchy is too complicated
- Lack of general functional structures like Monad and Functor. Though Scalaz provides most of needed classes.
- Invariance, also called deep immutability, cannot be typechecked.
- Function purity cannot be typechecked.
- There are way too many ways to do everything.

Haskell sucks because

- Positively infatuated with short and arbitrary names (even symbols) for everything. Assembly and C programmers have nothing on Haskell ones

because your programs look more like math when names are short. This says good things about your coding conventions rather than bad things about math conventions, apparently.

- Lazy evaluation makes memory leaks extra fun to debug.
- The pedantic and hard-to-learn type system.
- `String`, `ByteString`, `Lazy ByteString`, `Text`, `Lazy Text`.
- Non-reusable record field names.
- Which extensions are enabled isn't obvious when you're reading a file.
- Hierarchical modules are really just flat modules with dots in their names.
- Haskell makes it very easy to write multiple blocks of code that are extensionally but not intensionally identical. This problem is acute with third-party libraries.
- Horrid debugging tools; most of the time you have to use `print` statements. Basic errors like "Non-exhaustive patterns in function `foo`" don't provide a stack trace.
- It's hard to predict your code's performance.
- GHC is the most popular compiler, and is becoming the only one. Which would be fine if it weren't getting slower and its garbage collector were concurrent.

Clojure sucks because

- Lisp syntax provides no way of seeing what's a function etc. - No visual distinction.
- It has a slow startup time which makes it next to impossible to write a decently performant command line program.
- `Conj` function has inconsistent behavior depending on provided type (appends to the tail of vectors and the head of lists).

Go sucks because

Core language

- Go supports the nil pointer. This is similar to C's `void *`, an enormous source of bugs. Since nil can represent any type, it completely subverts the type system.

```
func randomNumber() *int {  
    return nil  
}  
  
func main() {  
    a := 1  
    b := randomNumber()  
    fmt.Printf("%d\n", a+b) // Runtime panic due to nil pointer  
}
```

- Because strings are just slices of bytes, there is no simple way to index or slice a string if it contains non-ASCII characters. You have to cast the string to a slice of "runes" (who calls characters that?), and then index

that slice.

- Despite the above, Go has two competing types for representing text, `string` and `[]byte`. `strings` are far superior to `[]bytes`, because they give Unicode characters when iterated over, can be compared with `== > <`, can be conveniently concatenated with `+`, and can be used as map keys. But important standard interfaces like `io.Writer` use `[]byte`.
- Similarly, the `len` function on strings returns the number of bytes in the string, which is not necessarily the number of characters. To get a string's true length, you have to use the well-named but horribly verbose library function `utf8.RuneCountInString()`
- Even though Go does not require `break` at the end of every case, the `break` statement still breaks out of `switch` statements. The same logic can always be accomplished by other control structures, but you need to resort to a labelled `break` to exit a loop while inside a `switch`.
- Deleting the *n*th element from a slice sure doesn't look like deletion: `slice = append(slice[:n], slice[n+1:]...)`
- Indexing into a map with a key that doesn't exist silently returns the zero value for the type of the values. To determine whether that was an actual value, you need to return a second boolean variable.
- If you import a library or declare a variable, but do not use it, your program will not compile even if everything else is valid.
- Composing functions that return multiple types is a mess.
- Go's error type is simply an interface to a function returning a string.
- Go lacks pattern matching & abstract data types.
- Go lacks immutable variables.
- Go lacks generics.
- Go lacks exceptions, and instead uses error codes everywhere, leading to repetitive error-checking boilerplate:

```
if v, err := otherFunc(); err != nil {
    return nil, err
} else {
    return doSomethingWith(v), nil
}
```

- Actually, Go supports *one* kind of exception, but calls it a panic. You can catch an exception, but Go calls it recovering. You can write "finally" blocks that run whether the function is exited by an exception or normally, but Go calls them deferred functions, and they run in reverse order from how they're written.
- Go's `range` loop iterates over a collection, assigning its keys and/or values to some number of variables, but the number of variables allowed and their meanings depend on the type of the collection, which might not be spelled out in the source code:

```
d := loadData()

for i := range d {
    // if d is a channel, i is an element read from the channel
    // if d is a map, i is a map key
    // otherwise, i is an array/slice/string index
}
```



```

for i, j := range d {
    // if d is a channel, this is illegal!
    // (even though when receiving normally from a channel,
    // you can get a second boolean value indicating whether it is closed)
    // if d is a string, i is an index and j is a rune (not necessarily d[i])
    // otherwise, i is an array/slice index or map key and j is d[i]
}

```

- Go lets you give names to return values so you can implicitly return them. It also lets you redefine variables in inner scopes, shadowing the definitions in outer scopes. That can make for unreadable code on its own, but the interaction between these two language features is just bizarre:

```

func foo1(i bool) (r int) {
    if i {
        r := 12
        fmt.Println(r)
    }
    return // returns 0
}

func foo2(i bool) (r int) {
    if i {
        r = 12
        fmt.Println(r)
        return // returns 12
    }
    return // returns 0
}

func foo3(i bool) (r int) {
    if i {
        r := 12
        fmt.Println(r)
        return // ERROR: r is shadowed during return
    }
    return
}

```

- Deferred functions can write to a function's named return values, leading to surprises:

```

func dogma() (i int) {
    defer func() {
        i++
    }()
    return 2 + 2
}

func main() {
    fmt.Println(dogma()) // prints 5
}

```

- Comparing an interface with nil checks whether the *type* of the interface is nil, and not its value. Thus forms a trap that has snared every Go programmer:

```

func returnsError() error {

```

```
var p *MyError = nil
if bad() {
    p = ErrBad
}
return p // Will always return a non-nil error.
}
```

Concurrency

- If multiple channels are available to receive from or send to, the `select` statement picks a case at random, meaning that to prioritize one channel over another you have to write:

```
select {
case <-chan1:
    doSomething()
default:
    select {
    case <-chan1:
        doSomething()
    case <-chan2:
        doSomethingElse()
    }
}
```

- The `select` statement is implemented as about 700 lines of runtime code (<https://github.com/golang/go/blob/master/src/runtime/select.go>). You can almost feel the performance decrease every time you use one.
- Goroutine leaks, where a goroutine loses all its ways of communicating with others, can leak whole *stacks* of memory that can't be garbage-collected.

Standard library

- Date/time format strings don't use the type of mnemonic codes seen in other languages, like "ddd" for abbreviated day or "%H" for hour. Instead, Go's library uses a system where "1" means month, "5" means seconds, "15" means hour, "6" means year, and so on. The documentation explains it in terms of a magic reference time (Mon Jan 2 15:04:05 MST 2006) and says "To define your own format, write down what the reference time would look like formatted your way". But it doesn't actually figure out how to transform times to meet your specifications; it simply recognizes a fixed, undocumented set of codes.
- Two implementations of random numbers, `math/rand` and `crypto/rand`.
- The `flag` package, which implements command-line flags, is not POSIX-compliant and doesn't support shorthand flags.
- The `errors` package is twenty lines long, because Go's error type is simply an interface to a function returning a string.

Toolchain

- Go's native package system does not support specifying versions or commits in the dependency information. Instead, the go community

recommends that each major release have its own separate repository; github.com/user/package/package-{v1,v2,v3}.

- Go's native package system does not support dependencies, forcing the community to create a lot of alternatives. (<https://github.com/golang/go/wiki/PackageManagementTools>)
- Version 1.5 converted the entire toolchain from C to Go. Because that was mostly done automatically, it caused significant performance decreases at both compile time and runtime.
- The compiler has an option documented as "reject unsafe code". All it does is prevent you from importing the `unsafe` package or any package that imports it recursively, which includes virtually the entire standard library, making any program you can compile with that option incapable of output and thus useless.
- The linker also has such an option, only it prevents compiling *any* program, because the Go runtime depends on `unsafe`.

Community

- Most complaints, especially about error handling, are brushed off as "you don't understand the language well enough".
- The contributors don't listen to the community.
 - Take this proposal to add aliases (<https://github.com/golang/go/issues/16339>), which was pushed through despite lots of negative feedback. (<https://groups.google.com/forum/#!topic/golang-dev/OmjsXkyOQpQ>) And when it was rolled back, it wasn't because the community asked for it (<https://github.com/golang/go/issues/16339#issuecomment-258531805>), but because there were problems even the contributors hadn't anticipated.
 - Or take this feature request (<https://github.com/golang/lint/issues/65>), which was immediately closed as WONTFIX despite significant community demand (though it was later implemented).
- Almost nothing in this list can be fixed, because of the Go 1 compatibility promise (<https://golang.org/doc/go1compat>). Until Go 2.0, that is, but that may never happen (https://docs.google.com/presentation/d/1JsCKdK_AvDdn8EkummMNvpo7ntqteWQfynq9hFTCkhQ/view?slide=id.p#slide=id.p).

Rust sucks because

Safety

- Borrowing rules are more strict than what's possible to safely do.
- The rules of `unsafe` are not strictly defined.
- LLVM's optimizer considers undefined behavior a license to kill. Of course, that only matters in `unsafe` code, but you need `unsafe` code for anything complicated.

API design and the type system

- Overly terse named types and keywords that don't communicate their purpose, like `Vec` and `Cell`.
- Rust has two main string types and four other string types, for a total of six. There are `String` and its slice equivalent `str` ("native" Rust UTF-8 string types used most of the time); `CString` and `cstr` (when compatibility with C is required); `OsString` and `osstr` (when working with the OS's `String`).
- Not smart enough coercion means that sometimes, you must use things like `&*some_var` (which will convert a smart pointer to a reference).
- You cannot use non-Sized types (like `str` and `[T]`) in several places, because every generic requires `Sized` unless you opt out by requiring `?Sized`.

Toolchain

- `rustc` is slow.
- Because it statically links everything, you get outdated copies of several libraries on your computer.
- Actually, it statically links *almost* everything. It dynamically links your program to `libc` (unless you target `musl` (<https://www.musl-libc.org/>), an alternative `libc`), so your executables aren't really self-contained.
- Modifying a file in your project or updating a dependency requires you to recompile everything that depends on it.
- Every executable, by default, contains a copy of `jemalloc`, making "hello world" approximately 650 kilobytes in size (<https://lifthrasiir.github.io/rustlog/why-is-a-rust-executable-large.html>).
- Type-ahead auto-completion is still a work in progress, because `rustc` is slow.
- IDE support is lacking.
- Generic types are very popular, and they're essentially copy-and-pasted for every concrete type they're used with. Compiling essentially the same code over and over again is painful when `rustc` is slow.
- The optimizer will break your program (<https://github.com/rust-lang/rust/issues/28728>) and it will run dog slow if you turn it off.
- Error messages from nested macros are difficult to understand.

Retrieved from "<https://wiki.theory.org/index.php?title=YourLanguageSucks&oldid=3521>"

-
- This page was last modified on 13 May 2017, at 15:43.
 - Content is available under Creative Commons Zero (Public Domain) unless otherwise noted.