Programming Skulpt

If you are reading this document, chances are you have used Skulpt in some form or another, maybe on skulpt.org or some other website. Or maybe you have embedded Skulpt on your own website. But, Skulpt is not complete. Bits and pieces of the Python language are missing, and now one of them is causing you enough pain that you have decided that you want to extend Skulpt with that missing bit. Or maybe you are just interested in learning a bit more about Skulpt and now you have found this document. Congratulations, thanks, and welcome.

What is Skulpt?

Skulpt is a system that compiles Python (of the 2.6-ish variety) into Javascript. But it's not Javascript that you can paste in to your browser and run. Python and Javascript are very different languanges, their types are different, their scoping rules are different. Python is designed to be run on Linux, or Windows, or Mac OS X, not in the browser! So, to provide a True Python experience Skulpt must provide a runtime environment in which the compiled code executes. This runtime environment is provided by the skulpt.min.js and skulpt-stdlib.js files that you must include in your web page in order to make Skulpt work.

To give you some idea of what is going on behind the scenes with skulpt lets look at what happens when our friend "hello world" is compiled from Python to Skulpt. We will revisit this program later and go into more detail, so for now, don't get bogged down in the detail, just have a look to see how much is really happening

Python Version

```
print "hello world"
```

Javascript Translation

```
/*     1 */ var $scope0 = (function($modname) {
/*     2 */     var $blk = 0,
/*     3 */         $exc = [],
/*     4 */         $gbl = {},
/*     5 */         $loc = $gbl,
/*     6 */         $err = undefined;
/*     7 */     $gbl.__name__ = $modname;
/*     8 */     Sk.globals = $gbl;
/*     9 */     try {
/*    10 */         while (true) {
/*    11 */             try {
/*    12 */                 switch ($blk) {
/*    13 */                 case 0:
/*    14 */                     /* --- module entry --- */
/*    15 */                     //
/*    16 */                     // line 1:
/*    17 */                     // print "hello world"
/*    18 */                     // ^
/*    19 */                     //
/*    20 */                     Sk.currLineNo = 1;
/*    21 */                     Sk.currColNo = 0
/*    22 */
/*    23 */
/*    24 */                     Sk.currFilename = './simple.py';
/*    25 */
/*    26 */                     var $str1 = new Sk.builtins['str']('hello world');
/*    27 */                     Sk.misceval.print_(new Sk.builtins['str']($str1).v);
```

```
/*    28 */                     Sk.misceval.print_("\n");
/*    29 */                     return $loc;
/*    30 */                     throw new Sk.builtin.SystemError('internal error: unterminat
/*    31 */                 }
/*    32 */             } catch (err) {
/*    33 */                 if ($exc.length > 0) {
/*    34 */                     $err = err;
/*    35 */                     $blk = $exc.pop();
/*    36 */                     continue;
/*    37 */                 } else {
/*    38 */                     throw err;
/*    39 */                 }
/*    40 */             }
/*    41 */         }
/*    42 */     } catch (err) {
/*    43 */         if (err instanceof Sk.builtin.SystemExit && !Sk.throwSystemExit) {
/*    44 */             Sk.misceval.print_(err.toString() + '\n');
/*    45 */             return $loc;
/*    46 */         } else {
/*    47 */             throw err;
/*    48 */         }
/*    49 */     }
/*    50 */ });
```

So, 50 lines of Javascript for hello world eh? That sounds kind of crazy, but you have to recognize that the environment with global variables, local variables, error handling, etc all has to happen even for the simplest program to run. The parts of the program above that really print "hello world" are lines 26-29. If you have a look at them you will see that we have to construct a string object from the string literal and then pass that off to some print function.

In the example above `Sk.builtin.str` and `Sk.misceval.print_` are part of the Skulpt runtime. It is usually the case that to extend Skulpt one of these runtime functions must be modified, or a new runtime function must be created and exposed so that it can be used in an ordinary Python program. The rest of this manual will take you through the essential parts of Skulpt so you can feel comfortable working on and extending the runtime environment.

An important thing to keep in mind as you are trying to understand Skulpt is that it is heavily influenced by the implementation of CPython. So although Python and Javascript are both object oriented languages many parts of the skulpt implementation are quite procedural. For example using functions that take an object as their first parameter may seem strange as we should have just created a method on that object. But in order to follow the CPython implementation this decision was made early on.

The Source

The `src` directory contains the javascript that implements skulpt as well as parts of the standard library. library modules are in src/lib. The source files could roughly be divided into two pieces. The compiler and the runtime. The compiler files are: `ast.js, parser.js, symtable.js, compile.js, and tokenize.js`The compiler part of skulpt reads python code and generates a Javascript program. If you want to change the syntax of Python these are the files to look at. The syntax used in skulpt is taken right from the Python 2.6.5 distribution.

When you run the program in the browser the javascript part is 'evaled' by javascript. The runtime files roughly correspond to all of the major object types in Python plus builtins:

- abstract.js -- contains lots of abstract function defs
- biginteger.js -- implements Python's long integer type
- bool.js
- skulpt-stdlib.js -- builtin functions: range, min, max, etc. are defined here

- builtindict.js -- Provides a mapping from the standard Python name to the internal name in skulpt-stdlib.js
- dict.js
- enumerate.js
- env.js
- errors.js -- Exceptions are defined here
- file.js
- float.js
- function.js
- generator.js
- import.js
- int.js
- list.js
- long.js
- method.js
- module.js
- native.js
- number.js
- object.js -- most things "inherit" from object
- set.js
- slice.js
- str.js
- timsort.js
- tuple.js
- type.js

Types and Namespaces

The `Sk` object contains all of the core Skulpt objects and functions. It's pretty easy to get from Sk.blah to its source. Usually you will see something like `Sk.builtin.foo` which indicates that you will likely find a corresponding file for foo in the src directory. Similarly `Sk.misceval.callsim` tells you that you should look in `misceval.js` for the callsim function.

Perhaps one of the most important concepts to learn when starting to program Skulpt is that you are always moving back and forth between Python objects and Javascript objects. Much of your job as a skulpt hacker is to either create Python objects as part of a builtin or module function, or interact with objects that have been created by the users "regular" Python code. Knowing when you are working with what is critical. For example a Javascript string is not the same thing as a python string. A Python string is really an instance of `Sk.builtin.str`and a Javscript string is an instance of `string`. You can't compare the two directly, and you definitely cannot use them interchangeably.

Python	Skulpt	Javascript
int	Sk.builtin.int	number
float	Sk.builtin.float	number
long	Sk.builtin.lng	NA
complex	Sk.builtin.complex	NA
list	Sk.builtin.list	Array
dict	Sk.builtin.dict	Object
set	Sk.builtin.set	NA
bool	Sk.builtin.bool	bool
tuple	Sk.builtin.tuple	NA

So how do I get the equivalent value? How do I work with these Python objects from Javascript?

There are two key functions in Sk.ffi: `Sk.ffi.remapToJs` and `Sk.ffi.remapToPy` These utility functions are smart enough to remap most builtin data types back and forth. So if you have a Python string and want to compare it to a Javascript string literal you just need to do `Sk.ffi.remapToJs(pystring)` to get a Javscript string you can compare.

If the Python object in question is a collection, remapToJs will work recursively and not only remap the top level object but also all of the contained objects.

When would you want to convert from Javascript to Python? Very often, in your implementation you will calculate a value that you want to return. The returned value needs to be a valid Python type. So lets say you calculate the factorial of a number in a new function you are adding to math. Then the resulting Javascript number must be turned into a Python object using `Sk.ffi.remapToPy(myresult)`.

In many places in the current codebase you will see the use of `somePythonObject.v` Where `v` is the actual javascript value hidden away inside the Python object. This is not the preferred way to obtain the mapping. Use the `Sk.ffi` API.

Skulpt is divided into several namespaces, you have already seen a couple of them, so here is the list

- Sk.abstr -- To extend skulpt you should know these functions
- Sk.builtin -- This is a big namespace that roughly corresponds to the Python `__builtin__` namespace
- Sk.ffi -- This is the foreign function interface. Good for mapping back and forth from Python to Javascript
- Sk.misceval -- To extend skulpt you should know these functions

The Generated Code

Perhaps one of the most instructive things you can do to understand Skulpt and how the pieces begin to fit together is to look at a simple Python program, and its translation to Javscript. So lets begin with Hello World.

Python Version

```
print "hello world"
```

Javascript Translation

```
/*     1 */ var $scope0 = (function($modname) {
/*     2 */     var $blk = 0,
/*     3 */         $exc = [],
/*     4 */         $gbl = {},
/*     5 */         $loc = $gbl,
/*     6 */         $err = undefined;
/*     7 */     $gbl.__name__ = $modname;
/*     8 */     Sk.globals = $gbl;
/*     9 */     try {
/*    10 */         while (true) {
/*    11 */             try {
/*    12 */                 switch ($blk) {
/*    13 */                 case 0:
/*    14 */                     /* --- module entry --- */
/*    15 */                     //
/*    16 */                     // line 1:
/*    17 */                     // print "hello world"
/*    18 */                     // ^
/*    19 */                     //
```

```
/*    20 */                    Sk.currLineNo = 1;
/*    21 */                    Sk.currColNo = 0
/*    22 */
/*    23 */
/*    24 */                    Sk.currFilename = './simple.py';
/*    25 */
/*    26 */                    var $str1 = new Sk.builtins['str']('hello world');
/*    27 */                    Sk.misceval.print_(new Sk.builtins['str']($str1).v);
/*    28 */                    Sk.misceval.print_("\n");
/*    29 */                    return $loc;
/*    30 */                    throw new Sk.builtin.SystemError('internal error: unterminat
/*    31 */                }
/*    32 */            } catch (err) {
/*    33 */                if ($exc.length > 0) {
/*    34 */                    $err = err;
/*    35 */                    $blk = $exc.pop();
/*    36 */                    continue;
/*    37 */                } else {
/*    38 */                    throw err;
/*    39 */                }
/*    40 */            }
/*    41 */        }
/*    42 */    } catch (err) {
/*    43 */        if (err instanceof Sk.builtin.SystemExit && !Sk.throwSystemExit) {
/*    44 */            Sk.misceval.print_(err.toString() + '\n');
/*    45 */            return $loc;
/*    46 */        } else {
/*    47 */            throw err;
/*    48 */        }
/*    49 */    }
/*    50 */ });
```

So, one line of python becomes 50 lines of Javscript. Luckily lots of this is boiler plate that is the same for every program. One important convention is that variables that start with a $ are variables that are generated by the compiler. So, in the above example $scope0, $blk, $str1, etc are all generated by the compiler not by the Python program. Each line of the python program gets a corresponding entry in the Sk.currLineNo so that runtime error messages or exceptions can reference the line that caused them.

For now lets concentrate on the parts of the code that were generated specifically for our program. That would be lines 26-29 above.

- 26: The compiler creates a variable to hold the string literal "hello world" A Python version of the string literal is created by calling the constructor `Sk.builtins['str']`passing the javascript string literal.
- 27: The `Sk.misceval.print_` function is called. Here is an interesting part of the runtime. The code for Sk.misceval.print_ is below. The key line is `Sk.output(s.v)` `Sk.output` is configurable to be any function that the web developer might want to provide. For example you might write a function that takes a javascript string as a parameter and updates a pre element. Or you might simply write a function that calls alert. Notice that `print_` simply expects to get an object. It converts this object into a Python string object by once again calling the string constructor `Sk.builtin.str`. If you've been keeping close watch, this is actually the third time our string liter has undergone this transformation. Luckily the string constructor is smart enough to simply return its parameter if the parameter is already a Python string. You might logically ask why does the compiler emit a call on line 27 when the runtime function takes care of the same issue. Not sure, maybe this is an optimization.

```
Sk.misceval.print_ = function(x)   // this was function print(x)   not sure why...
{
    if (Sk.misceval.softspace_)
    {
        if (x !== "\n") Sk.output(' ');
```

```
        Sk.misceval.softspace_ = false;
    }
    var s = new Sk.builtin.str(x);
    Sk.output(s.v);
    var isspace = function(c)
    {
        return c === '\n' || c === '\t' || c === '\r';
    };
    if (s.v.length === 0 || !isspace(s.v[s.v.length - 1]) || s.v[s.v.length - 1] === ' ')
        Sk.misceval.softspace_ = true;
};
```

- 28: print always results in a newline. So do it.
- 29: done return. This gets us out of the while(true) loop.

Another Example Naming Conventions

Python

```
x = 1
y = 2
z = x + y
print z
```

Javascript

```
/*     1 */ var $scope0 = (function($modname) {
/*     2 */     var $blk = 0,
/*     3 */         $exc = [],
/*     4 */         $gbl = {},
/*     5 */         $loc = $gbl,
/*     6 */         $err = undefined;
/*     7 */     $gbl.__name__ = $modname;
/*     8 */     Sk.globals = $gbl;
/*     9 */     try {
/*    10 */         while (true) {
/*    11 */             try {
/*    12 */                 switch ($blk) {
/*    13 */                 case 0:
/*    14 */                     /* --- module entry --- */
/*    15 */                     //
/*    16 */                     // line 1:
/*    17 */                     // x = 1
/*    18 */                     // ^
/*    19 */                     //
/*    20 */                     Sk.currLineNo = 1;
/*    21 */                     Sk.currColNo = 0
/*    22 */
/*    23 */
/*    24 */                     Sk.currFilename = './simple.py';
/*    25 */
/*    26 */                     $loc.x = new Sk.builtin.nmber(1, 'int');
/*    27 */                     //
/*    28 */                     // line 2:
/*    29 */                     // y = 2
/*    30 */                     // ^
/*    31 */                     //
```

```
/*    32 */                     Sk.currLineNo = 2;
/*    33 */                     Sk.currColNo = 0
/*    34 */
/*    35 */
/*    36 */                     Sk.currFilename = './simple.py';
/*    37 */
/*    38 */                     $loc.y = new Sk.builtin.nmber(2, 'int');
/*    39 */                     //
/*    40 */                     // line 3:
/*    41 */                     // z = x + y
/*    42 */                     // ^
/*    43 */                     //
/*    44 */                     Sk.currLineNo = 3;
/*    45 */                     Sk.currColNo = 0
/*    46 */
/*    47 */
/*    48 */                     Sk.currFilename = './simple.py';
/*    49 */
/*    50 */                     var $loadname1 = $loc.x !== undefined ? $loc.x : Sk.misceval
/*    51 */                     var $loadname2 = $loc.y !== undefined ? $loc.y : Sk.misceval
/*    52 */                     var $binop3 = Sk.abstr.numberBinOp($loadname1, $loadname2, '
/*    53 */                     $loc.z = $binop3;
/*    54 */                     //
/*    55 */                     // line 4:
/*    56 */                     // print z
/*    57 */                     // ^
/*    58 */                     //
/*    59 */                     Sk.currLineNo = 4;
/*    60 */                     Sk.currColNo = 0
/*    61 */
/*    62 */
/*    63 */                     Sk.currFilename = './simple.py';
/*    64 */
/*    65 */                     var $loadname4 = $loc.z !== undefined ? $loc.z : Sk.misceval
/*    66 */                     Sk.misceval.print_(new Sk.builtins['str']($loadname4).v);
/*    67 */                     Sk.misceval.print_("\n");
/*    68 */                     return $loc;
/*    69 */                     throw new Sk.builtin.SystemError('internal error: unterminat
/*    70 */                 }
/*    71 */             } catch (err) {
/*    72 */                 if ($exc.length > 0) {
/*    73 */                     $err = err;
/*    74 */                     $blk = $exc.pop();
/*    75 */                     continue;
/*    76 */                 } else {
/*    77 */                     throw err;
/*    78 */                 }
/*    79 */             }
/*    80 */         }
/*    81 */     } catch (err) {
/*    82 */         if (err instanceof Sk.builtin.SystemExit && !Sk.throwSystemExit) {
/*    83 */             Sk.misceval.print_(err.toString() + '\n');
/*    84 */             return $loc;
/*    85 */         } else {
/*    86 */             throw err;
/*    87 */         }
/*    88 */     }
/*    89 */ });
```

So, here we create some local variables. x, y, do some math to create a third local variable z, and then print it. Line 26 illustrates creating a local variable x (stored as an attribute of $loc) `new Sk.builtin.nmber(1, 'int');` By now you can probably guess that `Sk.builtin.nmber` is a constructor that creates a Python number object that is of type int, and has the value of 1. The same thing happens for `y`.

Next, on lines 40 -- 53 we see what happens in an assignment statement. first we load the values of x and y into temporary variables $loadname1 and $loadname2. Why not just use $loc.x ?? Well, we need to use Python's scoping rules. If $loc.x is undefined then we should check the outer scope to see if it exists there. `Sk.misceval.loadname` If loadname does not find a name `x` or `y` it throws a NameError, and execution would abort. You can see where this works by changing the assignment statement to `z = x + t` to purposely cause the error. The compiler blindly first tries $loc.t and then again calls loadname, which in this case does abort with an error!

On lines 52 and 53 we perform the addition using`Sk.abstr.numberBinOp($loadname1, $loadname2, 'Add');`Note the abstract (see abstract.js) nature of `numberBinOp` -- two parameters for the operands, and one parameter `'Add'` that indicates the operator. Finally the temporary result returned by numberBinOp is stored in $loc.z. It's important to note that $loc.z contains a Python number object. Down in the bowels of numberBinOp, the javascript numeric values for x and y are retrieved and result of adding two javascript numbers is converted to the appropriate type of Python object.

Function Calls, Conditionals, and Loops

Oh my! so what is the deal with this while(true)/try/switch thing? To understand this we need a bit more complicated example, so lets look at a program that contains an if/else conditional. We'll see that we now have a much more interesting switch statement.

Without showing all of the generated code, lets consider a simple python program like the one below. There will be two scope functions generated by the compiler for this example. $scope0 is for the main program where foo is defined and there is an if statement. The second $scope1 is for when the foo function is actually called. The $scope1 while/switch combo contains four cases: 0, 1, 2, and 3. You can imagine this python code consisting of four blocks. The first block starts at the beginning and goes through the evaluation of the if condition. The second block is the if true block of the if. The third block is the else block of the if statement, and the final block is the rest of the program after the if/else is all done. You can verify this for yourself by putting this program into a file `simple.py` and running `./skulpt.py run simple.py` If you examine the output you will see that the `$blk` variable is manipulated to control which `case` is executed the next time through the while loop. Very clever! If Javascript had `goto statements` this would probably look a lot different.

```
# <--- $blk 0 starts


def foo(bar):
    print bar
    x = 2
    if x % 2 == 0: # <---- end of $blk 0
        foo("hello") # <---- $blk 3
    else:
       foo("goodbye") # <---- $blk 2
    # <--- $blk 1 end of if
```

When foo is called, it has its own scope $scope1 created and called using Sk.misceval.callsim.

How do I add Feature X or Fix bug Y

Probably the biggest hurdle in working with skulpt is, "where do I start?" So, let me take you through a recent scenario, that is pretty illustrative of how I go about doing development on Skulpt.

The question was "how do I add keyword parameters (cmp, key, and reverse)" to the builtin sorted function. This is pretty tricky as Javascript does not support keyword parameters so there is no real straightforward path. So start as follows:

```
x = [1,2,3]
print(sorted(x,reverse=True))
```

Now run this using `./skulpt.py run test.py` and you will get a compiled program. With a little bit of sleuthing you find:

```
/*    35 */                    // line 2:
/*    36 */                    // print(sorted(x,reverse=True))
/*    37 */                    // ^
/*    38 */                    //
/*    39 */                    Sk.currLineNo = 2;
/*    40 */                    Sk.currColNo = 0
/*    41 */
/*    42 */
/*    43 */                    Sk.currFilename = './sd.py';
/*    44 */
/*    45 */                    var $loadname8 = $loc.sorted !== undefined ? $loc.sorted : S
/*    46 */                    var $loadname9 = $loc.x !== undefined ? $loc.x : Sk.misceval
/*    47 */                    var $call10 = Sk.misceval.call($loadname8, undefined, undefi
```

Where the important thing is to notice how the call is formatted after it is compiled. The fourth parameter to `Sk.misceval.call`is `['reverse', Sk.builtin.bool.true$]` Now if you check the source for misceval, you will see that these parameters are passed on to the apply function. In the apply function you will see that there is an assertion that the fourth parameter should be empty. Ok, here's our starting point to add in what's needed to actually process these key value parameters successfully.

In the case of a bug fix, you would do a similar thing, except that the line where your get an exception is likely to be closer to helping you figure out your next steps.

HOW TO

This section is for providing specific examples, or documentation on how to do a specific task. Suggestions for additional tasks are welcome!

Default Parameters

How do I add a function with named parameters with default values?

The key to this is that as the author of either a builtin function, or a method in a module, you need to add some meta data to the function definition. Here's an example of how we added the named parameters to the `sorted` function.

```
Sk.builtin.sorted = function sorted(iterable, cmp, key, reverse) {

/* body of sorted here */
}
Sk.builtin.sorted.co_varnames = ['cmp', 'key', 'reverse'];
Sk.builtin.sorted.$defaults = [Sk.builtin.none, Sk.builtin.none, false];
Sk.builtin.sorted.co_numargs = 4;
```

kwargs

How do I add a function with **kwargs?

Again the idea comes down to adding some meta-data after the function is defined. Here is an example of adding **kwargs to a method in a module:

```
var plotk_f = function(kwa)
    {
        Sk.builtin.pyCheckArgsLen("plotk", arguments.length, 0, Infinity, true, false)
        args = new Sk.builtins['tuple'](Array.prototype.slice.call(arguments, 1)); /*vararg*
        kwargs = new Sk.builtins['dict'](kwa);

        return new Sk.builtins['tuple']([args, kwargs]);
    };
    plotk_f['co_kwargs'] = true;
    mod.plotk = new Sk.builtin.func(plotk_f);
```

Adding a Module

This section is from a blog post I made in 2011, slightly updated.

So, here's the deal. skulpt relies on two javascript files the first is skulpt.min.js and skulpt-stdlib.js A very minimal installation only uses skulpt.min.js, whereas if you want to use any modules they are in skulpt-stdlib.js. Looking around the distribution you will not immediately find skulpt.min.js because you need to build it. You get a sculpt.js file by using the skulpty.py script that comes with the distribution. running `./skulpt.py --help` will give you the full list of commands, but the two that you probably most care about are `npm run build` and `npm run docbi` The dist command builds both skulpt.min.js and skulpt-stdlib.js docbi builds skulpt-stdlib.js and puts a new copy of it in the doc/static directory. Lets begin with a quick tour of the source tree:

- src - contains the implementation of the Python interpreter
- src/lib - has the module implementations of webgl and goog. This is where turtle will live and any other modules I implement along the way.
- doc - This directory contains a google app engine application and is what you see on skulpt.org There are a couple of important files to check out in here. One of them is doc/static/env/editor.js This is the code that ties together the interactive editor on the home page with the skulpt interpreter and the codemirror editor. If you know how to build a google app engine app then this directory makes sense. One thing about the home page is that it is not set up to use any of the modules. The modules are used in the more advanced ide, which you can find in doc/ide/static. I'm going to tell you how to add modules to the simpler editor later in this article.
- test - this directory contains a bunch of files for testing the implementation in a batch mode. These tests are run whenever you run `npm run build`, or `npm test`.
- dist - This directory gets created and populated when you run the `npm run build` command. It contains the built and compressed versions of skulpt.min.js and skulpt-stdlib.js

To illustrate how to make use of modules, here's an extended version of my earlier hello world style example.

```
<html>
<head>
<script src="skulpt.min.js" type="text/javascript"></script>
<script src="skulpt-stdlib.js" type="text/javascript"></script>

</head>
```

```
<body>
<script type="text/javascript">
function outf(text) {
   var mypre = document.getElementById("output");
   mypre.innerHTML = mypre.innerHTML + text;
}

function builtinRead(x)
{
    if (Sk.builtinFiles === undefined || Sk.builtinFiles["files"][x] === undefined)
        throw "File not found: '" + x + "'";
    return Sk.builtinFiles["files"][x];
}

function runit() {
   var prog = document.getElementById("yourcode").value;
   var mypre = document.getElementById("output");
   mypre.innerHTML = '';
   Sk.configure({output:outf,
           read: builtinRead
              });
   try {
      Sk.importMainWithBody("<stdin>",false,prog);
   } catch (e) {
      alert(e);
   }
}
</script>
<h3>Try This</h3>
<form>
<textarea edit_id="eta_5" id="yourcode">
print "Hello World"
</textarea>
<button onclick="runit()" type="button">Run</button>
</form>

<pre id="output"></pre>

</body>
</html>
```

There are some important differences between this version and the non-module version. First off, the call to Sk.configure contains another key value pair which sets up a specialized read function. This is the function that is responsible for returning your module out of the large array of files that are contained in the skulpt-stdlib.js file. You will see that all of the modules are contained in this one file, stored in a big JSON structure. The extra key value pair is: read: builtinRead

The read function is just for loading modules and is called when you do an import statement of some kind. In this case the function accesses the variable builtinFiles which is created from the skulpt-stdlib.js file. The other difference, of course, is that you have to include skulpt-stdlib.js in your html file. Note that skulpt-stdlib.js must be included after skulpt.min.js

Now as far as the module itself goes, the easiest thing to do is to start your module in the src/lib directory. This way it will automatically get built and included in skulpt-stdlib.js. If you don't put it there then you are going to have to modify the skulpt.py script, specifically the docbi function in the skulpt.py script to include your module. Suppose that you want to have a module called bnm.test Here's what you have to do. First, you need to make a bnm directory under lib. In this directory you will need to have either __init__.py or __init__.js or bnm.js to stand in for the bnm module. There doesn't need to be anything in the file as long as it exists. This is just like CPython by the way. Then to make a test module you can either make a test directory and put

all your javascript code in __init__.js or you can simply create a test.js file in the bnm directory. Let's look at the test module.

```
var $builtinmodule = function(name)
{
    var mod = {};
    var myfact = function(n) {
        if(n < 1) {
            return 1;
        } else {
            return n * myfact(n-1);
        }
    }
    mod.fact = new Sk.builtin.func(function(a) {
        return myfact(a.v);  // extract the underlying JS value with .v
    });

    mod.Stack = Sk.misceval.buildClass(mod, function($gbl, $loc) {
        $loc.__init__ = new Sk.builtin.func(function(self) {
            self.stack = [];
        });
        $loc.push = new Sk.builtin.func(function(self,x) {
            self.stack.push(x);
        });
        $loc.pop = new Sk.builtin.func(function(self) {
            return self.stack.pop();
        });
    }, 'Stack', []);

    return mod;
}
```

All modules start out with the $var builtinmodule = statement. This test module exposes a single method to the outside world, called fact, There are a couple of key functions for building up a module. The Sk.builtin.func call for adding functions to your module, and the Sk.misceval.buildClass method. This test module defines a simple factorial function called fact, and a class called stack. Here's a simple Python program that exercises the module:

```
import bnm.test
print 'starting'
print bnm.test.fact(10)
x = bnm.test.Stack()
x.push(1)
x.push(2)
print x.pop()
print 'done'
```

It's not obvious, but the buildClass method takes four parameters: globals, func, name, bases It seems that you always pass the mod object itself as the globals parameter, the func parameter is a function that represents the class object, the Name is the external name of the class, and bases presumably would be if the class is inheriting from another class.

The Sk.builtin.func method creates a function. For module creation we typically only have to worry about the one parameter, func, which is the javascript implementation of our Python function. The method can also take a globals object and two closure objects. Look at the comments in function.js if you want more explanation of how the builtin.func method works.

Well, I think this should be enough to get you going. It's worth repeating, if you made it this far, don't forget to call `npm run docbi` or `npm run build` after you make changes in your module, it's easy to get into the

mode of thinking that the new javascript is automatically loaded. But skulpt-stdlib.js is not automatically rebuilt!

Importing/Using a module in another module

While working on the namedtuple factory in the collections module I needed to add code to make sure that the fields named in the named tuple did not duplicate python keywords. while I was looking around for a list of keywords I discovered that there already was a list of keywords in the keyword module. Why not use that? A couple of problems:

- How do you import a module into another module? Especially under the condition where you are writing a module in javascript and the module you want to include is a python module?
- How do you call a function that was imported from a python module? Here is the snippet that demonstrates

```
var keywds = Sk.importModule("keyword", false, false);

mod.namedtuple = function (name, fields) {
    var nm = Sk.ffi.remapToJs(name);
    // fields could be a string or a tuple or list of strings
    var flds = Sk.ffi.remapToJs(fields);

    if (typeof(flds) === 'string') {
        flds = flds.split(/\s+/);
    }
    // use the keyword module function iskeyword
    for (i = 0; i < flds.length; i++) {
        if (Sk.ffi.remapToJs(Sk.misceval.callsim(keywds.$d['iskeyword'],Sk.ffi.remapToPy(flds
            throw new Sk.builtin.ValueError("Type names and field names cannot be a keyword:
        }
    }
```

The importing part is easy: `Sk.importModule(name, dumpJS, canSuspend)`

The not-so-obvious part is the line:`Sk.ffi.remapToJs(Sk.misceval.callsim(keywds.$d['iskeyword'],Sk.ffi.remapToPy(flds[i])))`

Working inside out: We use `Sk.misceval.callsim` to call the python function `iskeyword` which we retrieve from the module's dictionary of methods `$d` Because we are calling a Python function we need to remap the parameter from a javascript string to a Python string object. Hence the `remapToPy`call in the parameter. Since `iskeyword` will return a Python bool object we need to remap that back to javscript for our if statement.

You can use a similar strategy for creating an instance of a class:

```
var io = Sk.importModule("io", false, false);
var stdin = Sk.misceval.callsim(io.$d["TextIOWrapper"]);
```

Seems like a lot of work to check for a keyword in an array. But knowing how to do this for much more complicated methods in other modules will pay off.

Debugging

How do I use the debugger in the browser to help me debug my code?

Easy, just add the statement: `debugger;` to your code. Now if you have the javscript deveoper tools open in the browser you will have it.

If you want to start the debugger from a python function that you have written you can also add a debugger statement

If you want to enable debugging generally for use with `debugbrowser` follow these handy instructions:

- I make a new test using ./m nrt
- then add a debugger; to the start of the statement athttps://github.com/skulpt/skulpt/blob/master/src/import.js#L179the line would like this: `finalcode += "\ndebugger;" + co.funcname + "(" + namestr + ");";`
- run `npm run debugbrowser` wait until all tests have run
- startup the developer tools cmd+alt+i on a mac or F12 on a PC in chrome that is
- run the test I added before and it stops right before you enter the compiled code!

Development Workflow

1. Make a fork of the repository on github. DO NOT simply clone http://github.com/bnmnetp/runestone. Make a Fork. If you don't know how to make a fork consult the documentation here:https://help.github.com/articles/fork-a-repo
2. Make a simple myabs.py file that contains a few lines of python that exercise the abs function. Say it looks like this:

```
print abs(-1.0)
print abs(24)
```

3. Now go edit the source. To implement abs you would edit the builtin.js file. Now abs is pretty easy to add, because you can just have our skulpt version of abs call Math.abs So here it is

```
Sk.builtin.abs = function abs(x)
{
    return Math.abs(x);
};
```

You are not done yet, because builtin functions also have to be declared in the builtindict.js object as follows:

```
Sk.builtins = {
'range': Sk.builtin.range,
'len': Sk.builtin.len,
'min': Sk.builtin.min,
'max': Sk.builtin.max,
'sum': Sk.builtin.sum,
'abs': Sk.builtin.abs,
...
}
```

Now you can test your modifications from the command line by running:

```
./skulpt.py run myabs.py


-----
print abs(-1.0)
print abs(24)
-----
/*     1 */ var $scope0 = (function($modname) {
/*     2 */     var $blk = 0,
/*     3 */         $exc = [],
/*     4 */         $gbl = {},
/*     5 */         $loc = $gbl;
```

```
/*     6 */     $gbl.__name__ = $modname;
/*     7 */     while (true) {
/*     8 */         try {
/*     9 */             switch ($blk) {
/*    10 */             case 0:
/*    11 */                 /* --- module entry --- */
/*    12 */                 //
/*    13 */                 // line 1:
/*    14 */                 // print abs(-1.0)
/*    15 */                 // ^
/*    16 */                 //
/*    17 */                 Sk.currLineNo = 1;
/*    18 */                 Sk.currColNo = 0
/*    19 */
/*    20 */
/*    21 */                 Sk.currFilename = './myabs.py';
/*    22 */
/*    23 */                 var $loadname1 = $loc.abs !== undefined ? $loc.abs : Sk.misceval
/*    24 */                 var $call2 = Sk.misceval.callsim($loadname1, Sk.numberFromStr('-
/*    25 */                 Sk.misceval.print_(new Sk.builtins['str']($call2).v);
/*    26 */                 Sk.misceval.print_("\n");
/*    27 */                 //
/*    28 */                 // line 2:
/*    29 */                 // print abs(24)
/*    30 */                 // ^
/*    31 */                 //
/*    32 */                 Sk.currLineNo = 2;
/*    33 */                 Sk.currColNo = 0
/*    34 */
/*    35 */
/*    36 */                 Sk.currFilename = './myabs.py';
/*    37 */
/*    38 */                 var $loadname3 = $loc.abs !== undefined ? $loc.abs : Sk.misceval
/*    39 */                 var $call4 = Sk.misceval.callsim($loadname3, Sk.numberFromStr('2
/*    40 */                 Sk.misceval.print_(new Sk.builtins['str']($call4).v);
/*    41 */                 Sk.misceval.print_("\n");
/*    42 */                 return $loc;
/*    43 */                 goog.asserts.fail('unterminated block');
/*    44 */             }
/*    45 */         } catch (err) {
/*    46 */             if ($exc.length > 0) {
/*    47 */                 $blk = $exc.pop();
/*    48 */                 continue;
/*    49 */             } else {
/*    50 */                 throw err;
/*    51 */             }
/*    52 */         }
/*    53 */     }
/*    54 */ });
1
24
```

This is all incredibly useful information.

First it demonstrates that your addition actually worked. You can see the output at the bottom. Second, you can see how skulpt 'compiled' your python program into its intermediate Javascript form. While this may not be all that helpful in this particular case it can be very very helpful in figuring out what skulpt is actually doing. Now you should run all of the unit tests to make sure you have broken anything else accidentally. This is really easy:

```
npm test
```

If any tests fail it will be obvious that they did, and you'll have to do some investigation to figure out why. At the time of this writing you should see:

```
run: 343/343 (+1 disabled)
closure: skipped
```

Once you are satisfied that your extension is working fine. You should add a test case to test/run see: New Tests for instructions. This way we will have a permanent test in the bank of test cases in order to check for any future regressions.

Finally make a pull request on github to have your new feature integrated into the master copy. I probably will not accept your pull request if you haven't done step 4.

Outside of your editor, your browser, and your wits, the main development tool for skulpt is the skulpt.py command (also linked to m for historical compatibility).

```
./skulpt.py --help
```

Usage:

skulpt.py \<command> [\<options>] [script.py]

Commands:

run Run a Python file using Skulpt test Run all test cases dist Build core and library distribution files docbi Build library distribution file only and copy to doc/static

regenparser Regenerate parser tests regenasttests Regen abstract symbol table tests regenruntests Regenerate runtime unit tests regensymtabtests Regenerate symbol table tests regentests Regenerate all of the above

help Display help information about Skulpt host Start a simple HTTP server for testing upload Run appcfg.py to upload doc to live GAE site doctest Run the GAE development server for doc testing nrt Generate a file for a new test case runopt Run a Python file optimized browser Run all tests in the browser shell Run a Python program but keep a shell open (like python -i) vfs Build a virtual file system to support Skulpt read tests

debugbrowser Debug in the browser -- open your javascript console

Options:

-q, --quiet Only output important information -s, --silent Do not output anything, besides errors -u, --uncompressed Makes uncompressed core distribution file for debugging -v, --verbose Make output more verbose [default] --version Returns the version string in Bower configuration file.

Options: : --version show program's version number and exit -h, --help show this help message and exit -q, --quiet -s, --silent -u, --uncompressed -v, --verbose Make output more verbose [default]

run

The command `./skulpt.py run foo.py` compiles and runs a Python program generating output similar to the examples shown in the previous section. This is very common for development. For example if you find a bug, that you can express in a small Python program you can start by running the program from the

command line and inspecting the generated code. Usually this will give you a pretty good idea where the bug might be.

test

Run all the unit tests.

dist

Build the distribution files for skulpt:

- skulpt.min.js -- This is a minified version of the core interpreter files.
- skulpt-stdlib.js -- This is an unminified version of library functions. This file may contain javascript that implements a module, such as turtle or math, or it may contain pure python.

Building on windows

Running `.\skulpt.cmd dist` on windows requires some extra work, because the tests check against the text output, things with line-endings tend to get icky.

We want to make use we checkout skulpt with `LF` line endings, which is not default on windows. You have to configure git and reset your working directory. Like this:

```
> git config core.autocrlf input
> git update-index --refresh
> git rm --cached -r .
> git reset --hard
```

Getting stack traces from an exception

Sk.builtin.Exception objects have a property called 'traceback'. This property contains an Array of objects with 'filename', 'lineno' and (optionally) 'colno' properties, each representing a stack frame. The array is ordered from innermost to outermost frame.

If an object that is not an instance of Sk.builtin.Exception is thrown from within a Skulpt function (typically as part of an external piece of Javascript), it is wrapped in an Sk.builtin.ExternalError. The original object thrown is stringified (so the exception can be manipulated in Python), but a reference to the original is also saved in the ExternalError's 'nativeError' property so it can be inspected from Javascript.