py.js Documentation

Release 0.6

Xavier Morel

Oct 25, 2017

Contents

`py.js` is a parser and evaluator of Python expressions, written in pure javascript.

`py.js` is not intended to implement a full Python interpreter, its specification document is the Python 2.7 Expressions spec (along with the lexical analysis part) as well as the Python builtins.

CHAPTER 1

Supported Python builtins

py.**type**(*object*)
Gets the class of a provided object, if possible.

> **Note:** currently doesn't work correctly when called on a class object, will return the class itself (also, classes don't currently have a type).

py.**type**(*name*, *bases*, *dict*)
Not exactly a builtin as this form is solely javascript-level (currently). Used to create new `py.js` types. See *Implementing a custom type* for its usage.

py.**None**

py.**True**

py.**False**

py.**NotImplemented**

class py.**object**
Base class for all types, even implicitly (if no bases are provided to `py.type()`)

class py.**bool**([*object*])

class py.**float**([*object*])

class py.**str**([*object*])

class py.**unicode**([*object*])

class py.**tuple**

class py.**list**

class py.**dict**

py.**len**(*object*)

py.**isinstance**(*object*, *type*)

py.**issubclass**(*type*, *other_type*)

class py.**classmethod**

CHAPTER 2

Implementing a custom type

To implement a custom python-level type, one can use the *py.type()* builtin. At the JS-level, it is a function with the same signature as the `type` builtin[1]. It returns a child type of its one base (or *py.object* if no base is provided).

The `dict` parameter to *py.type()* can contain any attribute, javascript-level or python-level: the default `__getattribute__` implementation will ensure they are converted to Python-level attributes if needed. Most methods are also wrapped and converted to *Python-level callable*, although there are a number of special cases:

- Most "magic methods" of the data model ("dunder" methods) remain javascript-level. See *the listing of magic methods and their signatures*. As a result, they do not respect the *Python calling conventions*
- The `toJSON` and `fromJSON` methods are special-cased to remain javascript-level and don't follow the *Python calling conventions*
- Functions which have been wrapped explicitly (via *py.PY_def*, *py.classmethod* or `py.staticmethod`) are associated to the class untouched. But due to their wrapper, they will use the *Python calling conventions* anyway

Python-level callable

Wrapped javascript function *or* the `__call__()` method itself follow the *Python calling conventions*. As a result, they can't (easily) be called directly from javascript code. Because `__new__()` and `__init__()` follow from `__call__()`, they also follow the *Python calling conventions*.

py.PY_call() should be used when interacting with them from javascript is necessary.

Because `__call__` follows the *Python calling conventions*, instantiating a `py.js` type from javascript requires using *py.PY_call()*.

[1] with the limitation that, because py.js builds its object model on top of javascript's, only one base is allowed.

Python calling conventions

The python-level arguments should be considered completely opaque, they should be interacted with through *py.PY_parseArgs()* (to extract python-level arguments to javascript implementation code) and *py.PY_call()* (to call *Python-level callable* from javascript code).

A callable following the *Python calling conventions* ***must*** return a `py.js` object, an error will be generated when failing to do so.

Magic methods

`py.js` doesn't support calling magic ("dunder") methods of the datamodel from Python code, and these methods remain javascript-level (they don't follow the *Python calling conventions*).

Here is a list of the understood datamodel methods, refer to the relevant Python documentation for their roles.

Basic customization

__hash__()

Returns String

__eq__(*other*)

The default implementation tests for identity

Parameters **other** – *py.object* to compare this object with

Returns *py.bool*

__ne__(*other*)

The default implementation calls *__eq__()* and reverses its result.

Parameters **other** – *py.object* to compare this object with

Returns *py.bool*

__lt__(*other*)

The default implementation simply returns *py.NotImplemented*.

Parameters **other** – *py.object* to compare this object with

Returns *py.bool*

__le__(*other*)

The default implementation simply returns *py.NotImplemented*.

Parameters **other** – *py.object* to compare this object with

Returns *py.bool*

__ge__(*other*)

The default implementation simply returns *py.NotImplemented*.

Parameters **other** – *py.object* to compare this object with

Returns *py.bool*

__gt__(*other*)

The default implementation simply returns *py.NotImplemented*.

Parameters **other** – *py.object* to compare this object with

Returns *py.bool*

__str__()
Simply calls *__unicode__()*. This method should not be overridden, *__unicode__()* should be overridden instead.

Returns *py.str*

__unicode__()

Returns *py.unicode*

__nonzero__()
The default implementation always returns *py.True*

Returns *py.bool*

Customizing attribute access

__getattribute__(*name*)

Parameters **name** (*String*) – name of the attribute, as a javascript string

Returns *py.object*

__getattr__(*name*)

Parameters **name** (*String*) – name of the attribute, as a javascript string

Returns *py.object*

__setattr__(*name*, *value*)

Parameters

- **name** (*String*) – name of the attribute, as a javascript string
- **value** – *py.object*

Implementing descriptors

__get__(*instance*)

Note: readable descriptors don't currently handle "owner classes"

Parameters **instance** – *py.object*

Returns *py.object*

__set__(*instance*, *value*)

Parameters

- **instance** – *py.object*
- **value** – *py.object*

Emulating Numeric Types

- Non-in-place binary numeric methods (e.g. `__add__`, `__mul__`, ...) should all be supported including reversed calls (in case the primary call is not available or returns *py.NotImplemented*). They take a single *py.object* parameter and return a single *py.object* parameter.
- Unary operator numeric methods are all supported:

  __pos__()

  Returns *py.object*

  __neg__()

  Returns *py.object*

  __invert__()

  Returns *py.object*

- For non-operator numeric methods, support is contingent on the corresponding *builtins* being implemented

Emulating container types

__len__()

Returns py.int

__getitem__(*name*)

Parameters **name** – *py.object*

Returns *py.object*

__setitem__(*name*, *value*)

Parameters

- **name** – *py.object*
- **value** – *py.object*

__iter__()

Returns *py.object*

__reversed__()

Returns *py.object*

__contains__(*other*)

Parameters **other** – *py.object*

Returns *py.bool*

CHAPTER 3

Utility functions for interacting with `py.js` objects

Essentially the `py.js` version of the Python C API, these functions are used to implement new `py.js` types or to interact with existing ones.

They are prefixed with `PY_`.

`py.`**`PY_parseArgs`**(*arguments*, *format*)

Arguments parser converting from the *user-defined calling conventions* to a JS object mapping argument names to values. It serves the same role as PyArg_ParseTupleAndKeywords.

```
var args = py.PY_parseArgs(
    arguments, ['foo', 'bar', ['baz', 3], ['qux', "foo"]]);
```

roughly corresponds to the argument spec:

```
def func(foo, bar, baz=3, qux="foo"):
    pass
```

Note: a significant difference is that "default values" will be re-evaluated at each call, since they are within the function.

Parameters

- **`arguments`** – array-like objects holding the args and kwargs passed to the callable, generally the `arguments` of the caller.
- **`format`** – mapping declaration to the actual arguments of the function. A javascript array composed of five possible types of elements:
  - The literal string `'*'` marks all following parameters as keyword-only, regardless of them having a default value or not[1]. Can only be present once in the parameters list.

[1] Python 2, which py.js currently implements, does not support Python-level keyword-only parameters (it can be done through the C-API), but it seemed neat and easy enough so there.

- A string prefixed by `*`, marks the positional variadic parameter for the function: gathers all provided positional arguments left and makes all following parameters keyword-only[2]. `*args` is incompatible with `*`.
- A string prefixed with `**`, marks the positional keyword variadic parameter for the function: gathers all provided keyword arguments left and closes the argslist. If present, this must be the last parameter of the format list.
- A string defines a required parameter, accessible positionally or through keyword
- A pair of `[String, py.object]` defines an optional parameter and its default value.

For simplicity, when not using optional parameters it is possible to use a simple string as the format (using space-separated elements). The string will be split on whitespace and processed as a normal format array.

Returns a javascript object mapping argument names to values

Raises `TypeError` if the provided arguments don't match the format

class `py.`**`PY_def`**(*fn*)

Type wrapping javascript functions into py.js callables. The wrapped function follows *the py.js calling conventions*

Parameters **`fn`** (*`Function`*) – the javascript function to wrap

Returns a callable py.js object

Object Protocol

`py.`**`PY_hasAttr`**(*o*, *attr_name*)

Returns `true` if `o` has the attribute `attr_name`, otherwise returns `false`. Equivalent to Python's `hasattr(o, attr_name)`

Parameters

- **`o`** – A *`py.object`*
- **`attr_name`** – a javascript `String`

Return type `Boolean`

`py.`**`PY_getAttr`**(*o*, *attr_name*)

Retrieve an attribute `attr_name` from the object `o`. Returns the attribute value on success, raises `AttributeError` on failure. Equivalent to the python expression `o.attr_name`.

Parameters

- **`o`** – A *`py.object`*
- **`attr_name`** – a javascript `String`

Returns A *`py.object`*

Raises `AttributeError`

`py.`**`PY_str`**(*o*)

Computes a string representation of `o`, returns the string representation. Equivalent to `str(o)`

Parameters **`o`** – A *`py.object`*

[2] due to this and contrary to Python 2, py.js allows arguments other than `**kwargs` to follow `*args`.

Returns *py.str*

py.**PY_isInstance**(*inst*, *cls*)
Returns `true` if `inst` is an instance of `cls`, `false` otherwise.

py.**PY_isSubclass**(*derived*, *cls*)
Returns `true` if `derived` is `cls` or a subclass thereof.

py.**PY_call**(*callable[, args][, kwargs]*)
Call an arbitrary python-level callable from javascript.

Parameters

- **callable** – A `py.js` callable object (broadly speaking, either a class or an object with a `__call__` method)
- **args** – javascript Array of *py.object*, used as positional arguments to `callable`
- **kwargs** – javascript Object mapping names to *py.object*, used as named arguments to `callable`

Returns nothing or *py.object*

py.**PY_isTrue**(*o*)
Returns `true` if the object is considered truthy, `false` otherwise. Equivalent to `bool(o)`.

Parameters **o** – A *py.object*

Return type Boolean

py.**PY_not**(*o*)
Inverse of *py.PY_isTrue()*.

py.**PY_size**(*o*)
If `o` is a sequence or mapping, returns its length. Otherwise, raises `TypeError`.

Parameters **o** – A *py.object*

Returns `Number`

Raises `TypeError` if the object doesn't have a length

py.**PY_getItem**(*o*, *key*)
Returns the element of `o` corresponding to the object `key`. This is equivalent to `o[key]`.

Parameters

- **o** – *py.object*
- **key** – *py.object*

Returns *py.object*

Raises `TypeError` if `o` does not support the operation, if `key` or the return value is not a *py.object*

py.**PY_setItem**(*o*, *key*, *v*)
Maps the object `key` to the value `v` in `o`. Equivalent to `o[key] = v`.

Parameters

- **o** – *py.object*
- **key** – *py.object*
- **v** – *py.object*

Raises `TypeError` if `o` does not support the operation, or if `key` or `v` are not *py.object*

Number Protocol

py.**PY_add**(*o1*, *o2*)
Returns the result of adding o1 and o2, equivalent to o1 + o2.

Parameters

- **o1** – *py.object*
- **o2** – *py.object*

Returns *py.object*

py.**PY_subtract**(*o1*, *o2*)
Returns the result of subtracting o2 from o1, equivalent to o1 - o2.

Parameters

- **o1** – *py.object*
- **o2** – *py.object*

Returns *py.object*

py.**PY_multiply**(*o1*, *o2*)
Returns the result of multiplying o1 by o2, equivalent to o1 * o2.

Parameters

- **o1** – *py.object*
- **o2** – *py.object*

Returns *py.object*

py.**PY_divide**(*o1*, *o2*)
Returns the result of dividing o1 by o2, equivalent to o1 / o2.

Parameters

- **o1** – *py.object*
- **o2** – *py.object*

Returns *py.object*

py.**PY_negative**(*o*)
Returns the negation of o, equivalent to -o.

Parameters **o** – *py.object*

Returns *py.object*

py.**PY_positive**(*o*)
Returns the "positive" of o, equivalent to +o.

Parameters **o** – *py.object*

Returns *py.object*

CHAPTER 4

Differences with Python

- `py.js` completely ignores old-style classes as well as their lookup details. All `py.js` types should be considered matching the behavior of new-style classes
- New types can only have a single base. This is due to `py.js` implementing its types on top of Javascript's, and javascript being a single-inheritance language.

  This may change if `py.js` ever reimplements its object model from scratch.
- Piggybacking on javascript's object model also means metaclasses are not available (`py.type()` is a function)
- A python-level function (created through `py.PY_def()`) set on a new type will not become a method, it'll remain a function.
- `py.PY_parseArgs()` supports keyword-only arguments (though it's a Python 3 feature)
- Because the underlying type is a javascript `String`, there currently is no difference between `py.str()` and `py.unicode()`. As a result, there also is no difference between `__str__()` and `__unicode__()`.

Unsupported features

These are Python features which are not supported at all in `py.js`, usually because they don't make sense or there is no way to support them

- The `__delattr__`, `__delete__` and `__delitem__`: as `py.js` only handles expressions and these are accessed via the `del` statement, there would be no way to call them.
- `__del__` the lack of cross-platform GC hook means there is no way to know when an object is deallocated.
- `__slots__` are not handled
- Dedicated (and deprecated) slicing special methods are unsupported

Missing features

These are Python features which are missing because they haven't been implemented yet:

- Class-binding of descriptors doesn't currently work.
- Instance and subclass checks can't be customized
- "poor" comparison methods (`__cmp__` and `__rcmp__`) are not supported and won't be falled-back to.
- `__coerce__` is currently supported
- Context managers are not currently supported
- Unbound methods are not supported, instance methods can only be accessed from instances.

CHAPTER 5

Usage

To evaluate a Python expression, simply call *py.eval()*. *py.eval()* takes a mandatory Python expression parameter, as a string, and an optional evaluation context (namespace for the expression's free variables), and returns a javascript value:

```
> py.eval("t in ('a', 'b', 'c') and foo", {t: 'c', foo: true});
true
```

If the expression needs to be repeatedly evaluated, or the result of the expression is needed in its "python" form without being converted back to javascript, you can use the underlying triplet of functions *py.tokenize()*, *py.parse()* and *py.evaluate()* directly.

CHAPTER 6

API

Core functions

py.**eval**(*expr*[, *context*])
"Do everything" function, to use for one-shot evaluation of Python expressions. Chains tokenizing, parsing and evaluating the expression then *converts the result back to javascript*

Parameters

- **expr** (*String*) – Python expression to evaluate
- **context** (*Object*) – evaluation context for the expression's free variables

Returns the expression's result, converted back to javascript

py.**tokenize**(*expr*)
Expression tokenizer

Parameters **expr** (*String*) – Python expression to tokenize

Returns token stream

py.**parse**(*tokens*)
Parses a token stream and returns the corresponding parse tree.

The parse tree is stateless and can be memoized and reused for frequently evaluated expressions.

Parameters **tokens** – token stream from *py.tokenize()*

Returns parse tree

py.**evaluate**(*tree*[, *context*])
Evaluates the expression represented by the provided parse tree, using the provided context for the exprssion's free variables.

Parameters

- **tree** – parse tree returned by *py.parse()*

- **context** – evaluation context

Returns the "python object" resulting from the expression's evaluation

Return type *py.object*

Conversions from Javascript to Python

`py.js` will automatically attempt to convert non-*py.object* values into their `py.js` equivalent in the following situations:

- Values passed through the context of *py.eval()* or *py.evaluate()*
- Attributes accessed directly on objects
- Values of mappings passed to *py.dict*

Notably, `py.js` will *not* attempt an automatic conversion of values returned by functions or methods, these must be *py.object* instances.

The automatic conversions performed by `py.js` are the following:

- `null` is converted to *py.None*
- `true` is converted to *py.True*
- `false` is converted to *py.False*
- numbers are converted to *py.float*
- strings are converted to *py.str*
- functions are wrapped into `py.PY_dev`
- `Array` instances are converted to *py.list*

The rest generates an error, except for `undefined` which specifically generates a `NameError`.

Conversions from Python to Javascript

py.js types (extensions of `py.object()`) can be converted back to javascript by calling their `py.object.toJSON()` method.

The default implementation raises an error, as arbitrary objects can not be converted back to javascript.

Most built-in objects provide a `py.object.toJSON()` implementation out of the box.

Javascript-level exceptions

Javascript allows throwing arbitrary things, but runtimes don't seem to provide any useful information (when they ever do) if what is thrown isn't a direct instance of `Error`. As a result, while `py.js` tries to match the exception-throwing semantics of Python it only ever throws bare `Error` at the javascript-level. Instead, it prefixes the error message with the name of the Python expression, a colon, a space, and the actual message.

For instance, where Python would throw `KeyError("'foo'")` when accessing an invalid key on a `dict`, `py.js` will throw `Error("KeyError: 'foo'")`.

Index

Symbols

P