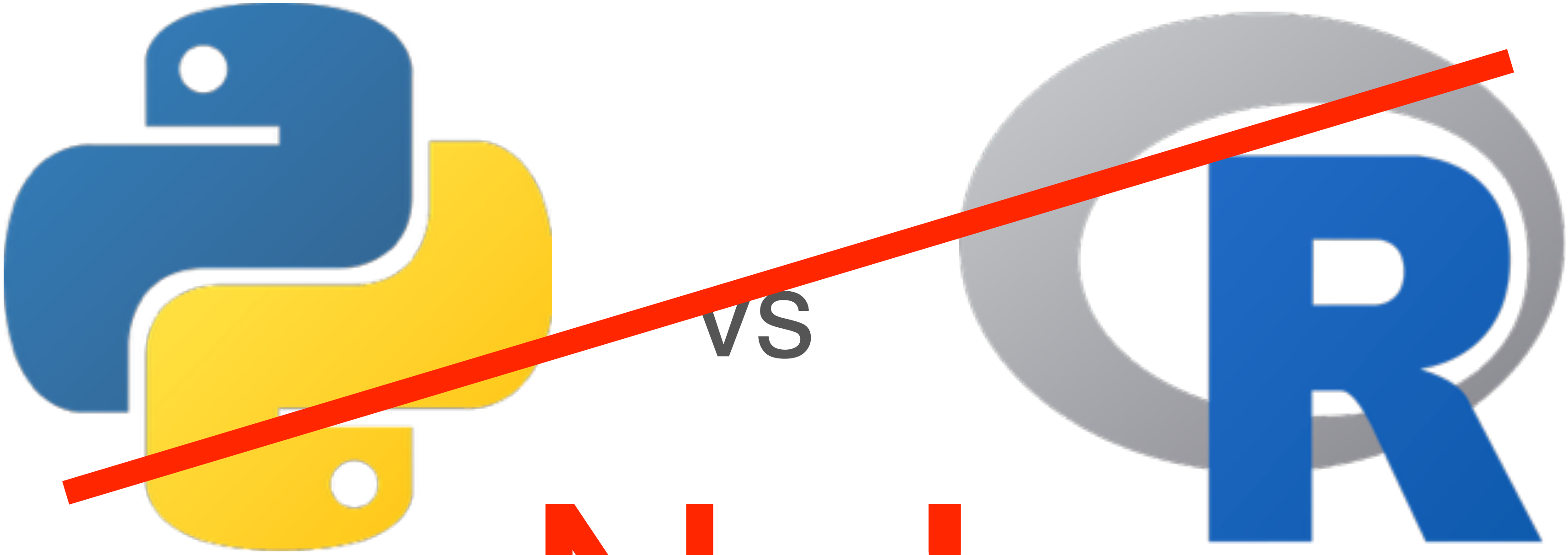
Python and R Together at Last

Writing Cross-Language Tools

CIVIS™
ANALYTICS

Building a Data-Driven World™

vs



~~vs~~

No!

Meet users where they are

Prior Knowledge

R is popular in some fields, Python in others. Diverse teams are often polyglot.

Availability of Key Packages

Important packages are often available in only one language. ***NLTK*** in Python, ***glmnet*** in R. This means a data science workflow often needs to use multiple languages.

Tradeoffs

Different languages optimize for different things. Python is a general purpose language, R is optimized for statistics/ manipulation of tabular data, Go is a great fit for network services.

Some tools are already cross-language

How?

Two Options

Native/Compiled Extensions (C/C++)

Pros

- fast!
- many languages speak C

Cons

- takes more code
- difficult

Examples

- Stan
- XGBoost

RPC over TCP/HTTP or IPC

Pros

- every language speaks TCP/HTTP
- easy to “wire up” host language

Cons

- cost of communication

Examples

- Spark
- H2o

Two Options

Native/Compiled Extensions (C/C++)

Pros

- fast!
- many languages speak C

Cons

- takes more code
- difficult

Examples

- Stan
- XGBoost

Our focus for today.

RPC over TCP/HTTP or IPC

Pros

- every language speaks TCP/HTTP
- easy to "wire up" host language
- cost of communication

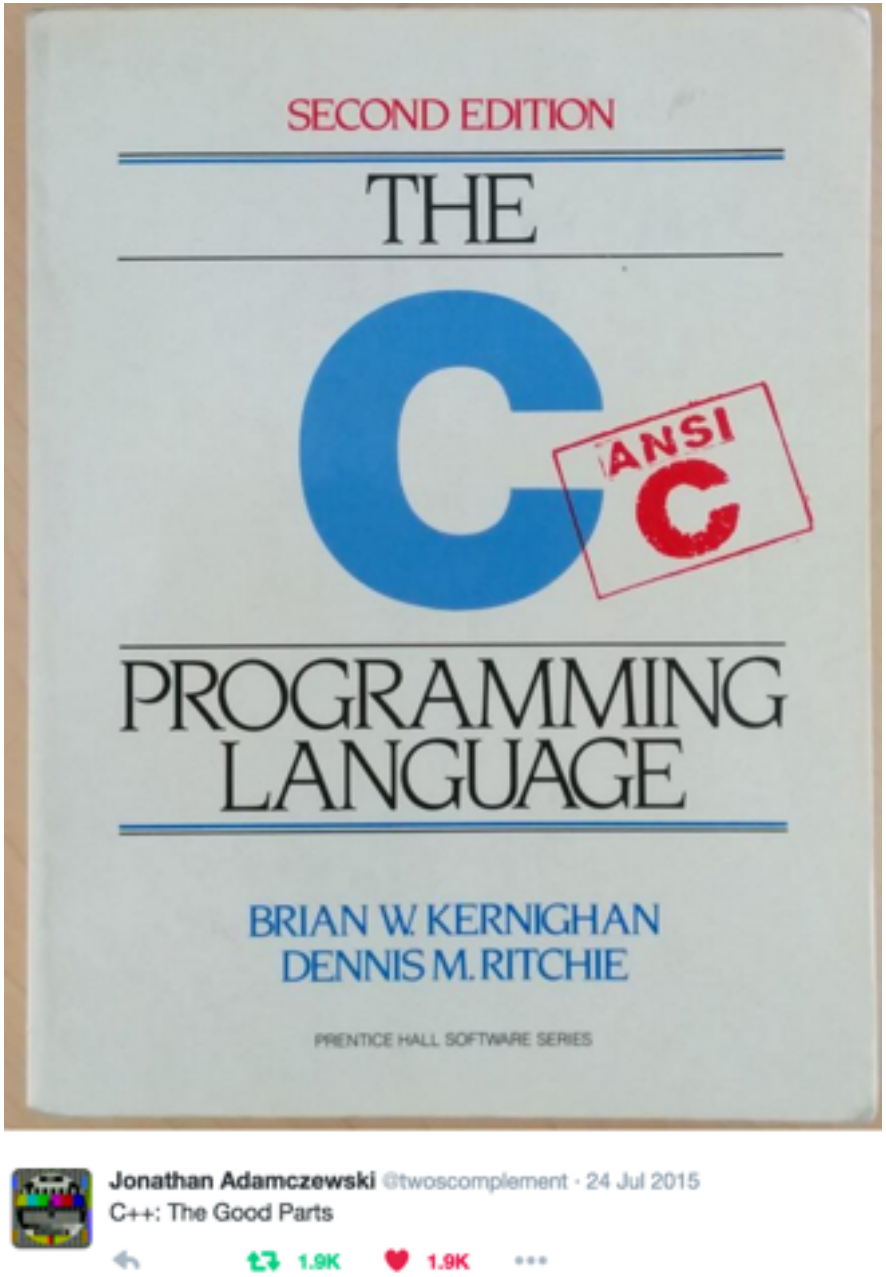
Examples

- Spark
- H2o

Why C

- Python and R “speak” C
- Fast!
- Portable (mostly)
- Simple

C++: The Good Parts



Jonathan Adamczewski @twoscomplement · 24 Jul 2015

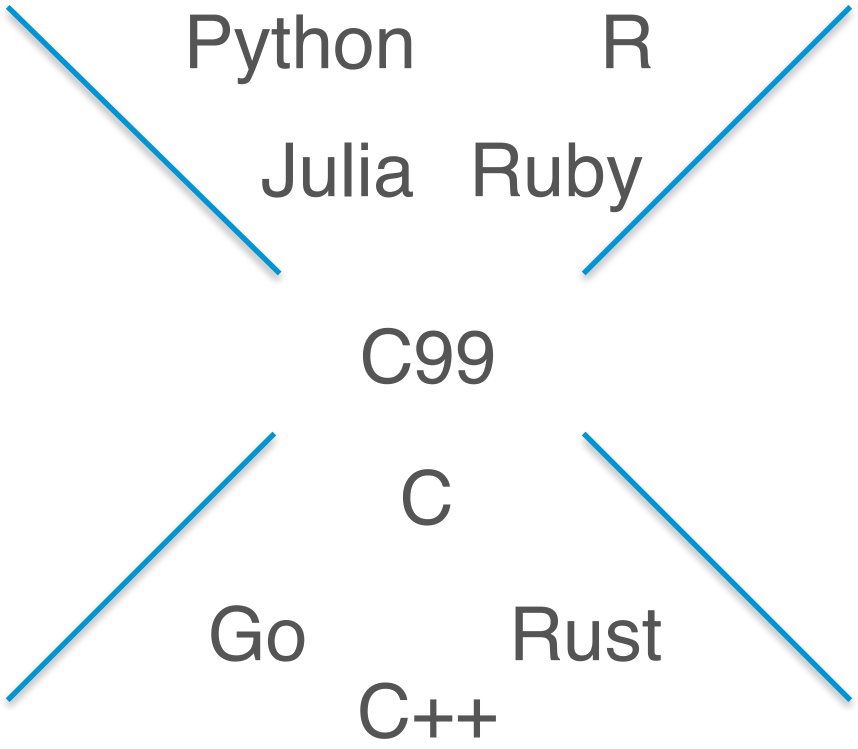
C++: The Good Parts

1.9K 1.9K

Modern C

- tooling has come a long way
- various “sanitizers”
  - address/memory sanitizer
  - undefined behavior sanitizer
  - leak sanitizer
  - thread sanitizer
- clang gives much better error messages

Alternatives: The Hourglass Interface



Python R

Julia Ruby

C99

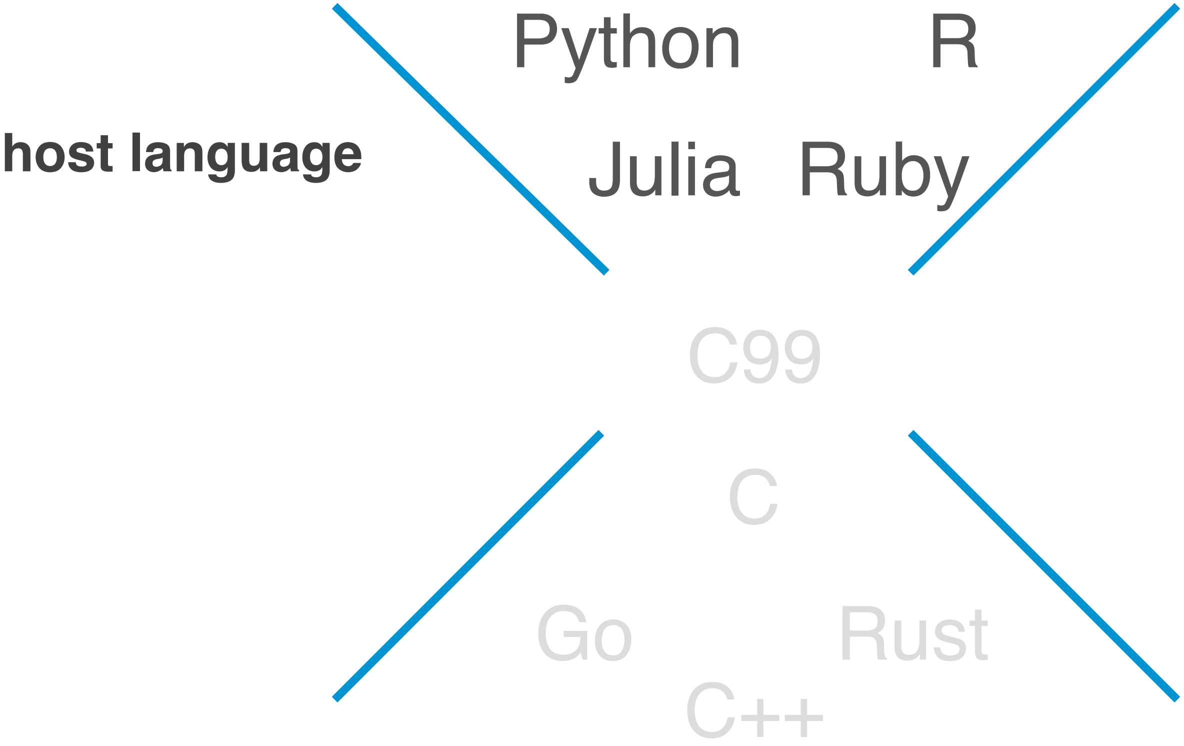
C

Go Rust

C++

Credit: Hourglass Interfaces for C++ APIs, Stefanus Du Toit

Alternatives: The Hourglass Interface



host language

Python R

Julia Ruby

C99

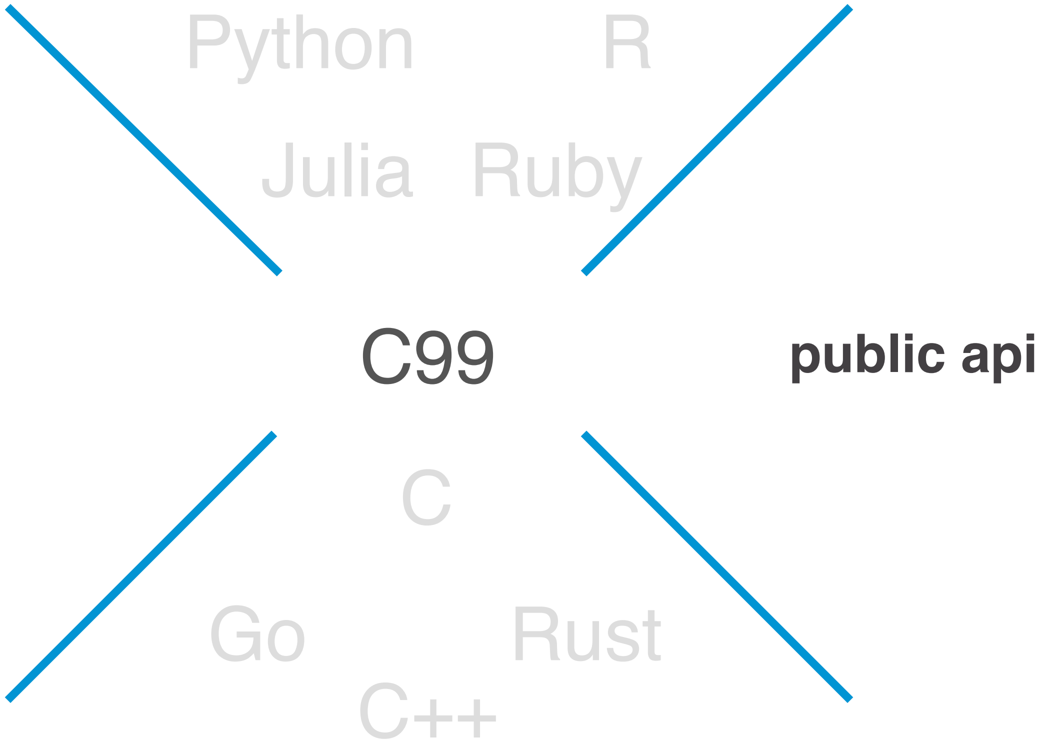
C

Go Rust

C++

Credit: Hourglass Interfaces for C++ APIs, Stefanus Du Toit

Alternatives: The Hourglass Interface



Python R

Julia Ruby

C99

public api

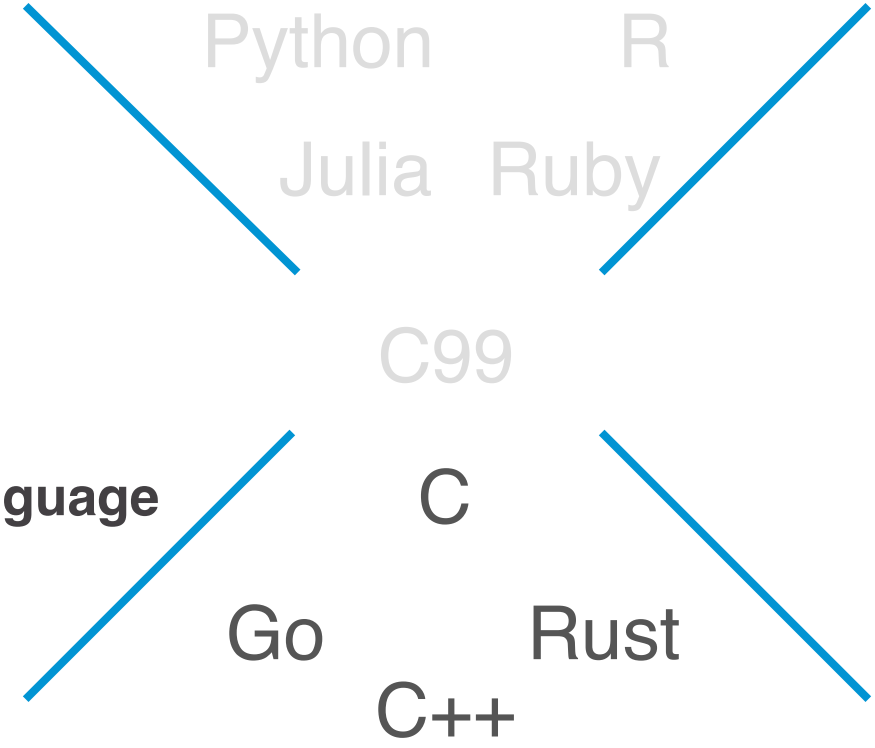
C

Go Rust

C++

Credit: Hourglass Interfaces for C++ APIs, Stefanus Du Toit

Alternatives: The Hourglass Interface



Python R

Julia Ruby

C99

implementation language

C

Go Rust

C++

Credit: Hourglass Interfaces for C++ APIs, Stefanus Du Toit

Example

The Mighty Summation Function

```python
def tally(s):
    total = 0
    for elm in s:
        total += elm
    return total
```

Note: It’s best to start development in a language like python.

Smoke Test

```
In [1]: tally([1, 2, 3])
Out[1]: 6
```

C Implementation

```c
#include <stddef.h>

double tally(double *s, size_t n) {
    double total = 0;
    for (size_t i = 0; i < n; i++) {
        total += s[i];
    }
    return total;
}
```

C Implementation

```c
#include <stddef.h>

double tally(double *s, size_t n) {
    double total = 0;
    for (size_t i = 0; i < n; i++) {
        total += s[i];

    return total;
}
```

need to pass the length

C/C++ and Python

- Cython
- CFFI
- ctypes
- C (via the Python C API)

The Python C API

```c
#include <stdio.h>
#include "Python.h"
#include "tally.h"

static PyObject *tally_(PyObject *self, PyObject *args) {
    // decode/cast the args
    // call our C function tally
    // build the result
}

// module method table
static PyMethodDef MethodTable[] = {
    // ...
};

// module def
static struct PyModuleDef tally_module = {
    // ...
};

// module init
PyMODINIT_FUNC PyInit_tally_py(void) {
    return PyModule_Create(&tally_module);
}
```

The Python C API: Buffer API

```c
static PyObject *tally_(PyObject *self, PyObject *args) {
    PyObject *buf;
    if (!PyArg_ParseTuple(args, "O", &buf)) {
        return NULL;
    }

    Py_buffer view;
    int buf_flags = PyBUF_ANY_CONTIGUOUS | PyBUF_FORMAT;
    if (PyObject_GetBuffer(buf, &view, buf_flags) == -1) {
        return NULL;
    }

    if (strcmp(view.format,"d") != 0) {
        PyErr_SetString(PyExc_TypeError, "we only take floats :(");
        PyBuffer_Release(&view);
        return NULL;
    }

    double result = tally(view.buf, view.shape[0]);
    PyBuffer_Release(&view);
    return Py_BuildValue("d", result);
}
```

The Python C API: Buffer API

```c
static PyObject *tally_(PyObject *self, PyObject *args) {
    PyObject *buf;
    if (!PyArg_ParseTuple(args, "O", &buf)) {
        return NULL;
    }

    Py_buffer view;
    int buf_flags = PyBUF_ANY_CONTIGUOUS | PyBUF_FORMAT;
    if (PyObject_GetBuffer(buf, &view, buf_flags) == -1) {
        return NULL;
    }

    if (strcmp(view.format,"d") != 0) {
        PyErr_SetString(PyExc_TypeError, "we only take floats :(");
        PyBuffer_Release(&view);
        return NULL;
    }

    double result = tally(view.buf, view.shape[0]);
    PyBuffer_Release(&view);
    return Py_BuildValue("d", result);
}
```

The Python C API: Buffer API

```c
static PyObject *tally_(PyObject *self, PyObject *args) {
    PyObject *buf;
    if (!PyArg_ParseTuple(args, "O", &buf)) {
        return NULL;
    }

    Py_buffer view;
    int buf_flags = PyBUF_ANY_CONTIGUOUS | PyBUF_FORMAT;
    if (PyObject_GetBuffer(buf, &view, buf_flags) == -1) {
        return NULL;
    }

    if (strcmp(view.format,"d") != 0) {
        PyErr_SetString(PyExc_TypeError, "we only take floats :(");
        PyBuffer_Release(&view);
        return NULL;
    }

    double result = tally(view.buf, view.shape[0]);
    PyBuffer_Release(&view);
    return Py_BuildValue("d", result);
}
```

The Python C API: Buffer API

```c
static PyObject *tally_(PyObject *self, PyObject *args) {
    PyObject *buf;
    if (!PyArg_ParseTuple(args, "O", &buf)) {
        return NULL;
    }

    Py_buffer view;
    int buf_flags = PyBUF_ANY_CONTIGUOUS | PyBUF_FORMAT;
    if (PyObject_GetBuffer(buf, &view, buf_flags) == -1) {
        return NULL;
    }

    if (strcmp(view.format,"d") != 0) {
        PyErr_SetString(PyExc_TypeError, "we only take floats :(");
        PyBuffer_Release(&view);
        return NULL;
    }

    double result = tally(view.buf, view.shape[0]);
    PyBuffer_Release(&view);
    return Py_BuildValue("d", result);
}
```

The Python C API

```c
static PyObject *tally_(PyObject *self, PyObject *args) {
    PyObject *buf;
    if (!PyArg_ParseTuple(args, "O", &buf)) {
        return NULL;
    }

    Py_buffer view;
    int buf_flags = PyBUF_ANY_CONTIGUOUS | PyBUF_FORMAT;
    if (PyObject_GetBuffer(buf, &view, buf_flags) == -1) {
        return NULL;
    }

    if (strcmp(view.format,"d") != 0) {
        PyErr_SetString(PyExc_TypeError, "we only take floats :(");
        PyBuffer_Release(&view);
        return NULL;
    }

    double result = tally(view.buf, view.shape[0]);
    PyBuffer_Release(&view);
    return Py_BuildValue("d", result);
}
```

The Python C API

```
static PyObject *tally_(PyObject *self, PyObject *args) {
    PyObject *buf;
    if (!PyArg_ParseTuple(args, "O", &buf)) {
        return NULL;
    }

    Py_buffer view;
    int buf_flags = PyBUF_ANY_CONTIGUOUS | PyBUF_FORMAT;
    if (PyObject_GetBuffer(buf, &view, buf_flags) == -1) {
        return NULL;
    }

    if (strcmp(view.format,"d") != 0) {
        PyErr_SetString(PyExc_TypeError, "we only take floats :(");
        PyBuffer_Release(&view);
        return NULL;
    }

    double result = tally(view.buf, view.shape[0]);
    PyBuffer_Release(&view);
    return Py_BuildValue("d", result);
}
```

The Python C API: Method Table

```c
static PyMethodDef MethodTable[] = {
    {"tally", &tally_, METH_VARARGS, "Compute the sum of an array."},
    { NULL, NULL, 0, NULL}
};

static struct PyModuleDef tally_module = {
    .m_base = PyModuleDef_HEAD_INIT,
    .m_name = "tally_py",
    .m_size = -1,
    .m_methods = MethodTable
};

PyMODINIT_FUNC PyInit_tally_py(void) {
    return PyModule_Create(&tally_module);
}
```

C/C++ and R

- Rcpp
- C (via the R C API)

The R C API

```c
#include <R.h>
#include <Rinternals.h>
#include <R_ext/Rdynload.h>
#include "tally.h"

SEXP tally_(SEXP x_) {
    // cast/decode the input
    // call our tally function
    // build the output
}

// method table
static R_CallMethodDef callMethods[] = {
    // ...
};

// module/package init
void R_init_tally_r(DllInfo *info) {
  R_registerRoutines(info, NULL, callMethods, NULL, NULL);
}
```

The R C API

```c
SEXP tally_(SEXP x_) {
  double *x = REAL(x_);
  int n = length(x_);

  SEXP out = PROTECT(allocVector(REALSXP, 1));
  REAL(out)[0] = tally(x, n);
  UNPROTECT(1);

  return out;
}
```

The R C API

```
SEXP tally_(SEXP x_) {
  double *x = REAL(x_);
  int n = length(x_);

  SEXP out = PROTECT(allocVector(REALSXP, 1));
  REAL(out)[0] = tally(x, n);
  UNPROTECT(1);

  return out;
}
```

The R C API

```c
SEXP tally_(SEXP x_) {
  double *x = REAL(x_);
  int n = length(x_);

  SEXP out = PROTECT(allocVector(REALSXP, 1));
  REAL(out)[0] = tally(x, n);
  UNPROTECT(1);

  return out;
}
```

The R C API

```
SEXP tally_(SEXP x_) {
  double *x = REAL(x_);
  int n = length(x_);

  SEXP out = PROTECT(allocVector(REALSXP, 1));
  REAL(out)[0] = tally(x, n);
  UNPROTECT(1);

  return out;
}
```

The R C API: Function Registration

```c
static R_CallMethodDef callMethods[] = {
  {"tally_", (DL_FUNC)&tally_, 1},
  {NULL, NULL, 0}
};

void R_init_tally_r(DllInfo *info) {
  R_registerRoutines(info, NULL, callMethods, NULL, NULL);
}
```

Miscellaneous

Dependencies

Don’t depend on APIs from host languages, i.e., numpy, rmath

Errors

Use error codes to signal problems. Don’t call abort or exit as these will quit the process running the host language.

Memory

Typically best to make the host language responsible for allocation and deallocation. It’s challenging to transfer ownership over the boarder.

Logging/Verbosity

At the very least, make this optional.

Compiler

Trust the compiler it’s smarter than all of us. Ensure your code compiles without warnings.

Parting Thoughts

1. Meet users where they are
2. Reach a larger audience
3. Make a bigger impact

Thank You

Bill Lattner
twitter: @wlattner
github: github.com/wlattner