present

Programming scientifically in Julia

- Julia is easy to read and write (but so are Python or Matlab)
- Julia is performant and scalable* (but so are C or Fortran)
- Julia is interactive (but so are Python or Mathematica)

> Researchers often find themselves coding algorithms in one programming language, only to have to rewrite them in a faster one.

Julia solves the two-language problem.

Power of (a good) Language

Abelman and Sussman, Structure and Interpretation of Computer Programs

> Programs must be written for people to read, and only incidentally for machines to execute.

Bertrand Russell

> Language serves not only to express thoughts, but to make possible thoughts which could not exist without it.

Ludwig Wittgenstein

> The limits of my language mean the limits of my world.

So why exactly is Julia great?

Julia is easy to learn

This easiness arrises from a clear and concise syntax (similar but quite more refined than Python), which results in small but clear code.

So why exactly is Julia great?

Julia is (easily) fast

Tricky definition:

- Most languages are "equivalent" on a theoretical level but in practise greatly differ
- One can write terribly slow programs in languages that are supposed to be fast

The norm is that something in pure Julia will run a lot faster than in most dynamic languages.

Native Julia can beat highly optimised libraries (some of the fastest BLAS-like operations are in `LoopVectorization.jl`). No magic: Julia shares the same LLVM backend with e.g. C++ so the compiled performances can be very similar.

Don't focus on the fact that Julia is faster or not but **that it's simply much simpler to write high-performant and clear code in Julia than in other languages**.

So why exactly is Julia great?

Julia is (easily) fast

Calculate the mean of an array ignoring NaNs **in Julia**

```
function mean_count(x::AbstractArray{T}) where T<:AbstractFloat
    z = zero(eltype(x))
    sum = z
    count = 0
    @simd for i in x
        count += ifelse(isnan(i), 0, 1)
        sum += ifelse(isnan(i), z, i)
    end
```

```
    result = sum / count
    return (result, count)
end
```

So why exactly is Julia great?

Julia is (easily) fast

Calculate the mean of an array ignoring NaNs **in Python (Numpy)**

```
/* dtype = [['float64'], ['float32']] */
REDUCE_ALL(nanmean, DTYPE0) {
    Py_ssize_t count = 0;
    npy_DTYPE0 ai, asum = 0;
    INIT_ALL
    BN_BEGIN_ALLOW_THREADS
    WHILE {
        FOR {
            ai = AI(DTYPE0);
            if (!bn_isnan(ai)) {
                asum += ai;
                count += 1;
            }
        }
        NEXT
    }
    BN_END_ALLOW_THREADS
    if (count > 0) {
        return PyFloat_FromDouble(asum / count);
    } else {
        return PyFloat_FromDouble(BN_NAN);
    }
}

REDUCE_ONE(nanmean, DTYPE0) {
    Py_ssize_t count;
    npy_DTYPE0 ai, asum;
    INIT_ONE(DTYPE0, DTYPE0)
    BN_BEGIN_ALLOW_THREADS
    if (LENGTH == 0) {
        FILL_Y(BN_NAN)
```

```
    } else {
        WHILE {
            count = 0;
            asum = 0;
            FOR {
                ai = AI(DTYPE0);
                if (!bn_isnan(ai)) {
                    asum += ai;
                    count += 1;
                }
            }
            if (count > 0) {
                asum /= count;
            } else {
                asum = BN_NAN;
            }
            YPP = asum;
            NEXT
        }
    }
    BN_END_ALLOW_THREADS
    return y;
}
/* dtype end */

/* dtype = [['int64', 'float64'], ['int32', 'float64']] */
REDUCE_ALL(nanmean, DTYPE0) {
    Py_ssize_t total_length = 0;
    npy_DTYPE1 asum = 0;
    INIT_ALL
    BN_BEGIN_ALLOW_THREADS
    WHILE {
        FOR asum += AI(DTYPE0);
        total_length += LENGTH;
        NEXT
    }
    BN_END_ALLOW_THREADS
    if (total_length > 0) {
        return PyFloat_FromDouble(asum / total_length);
    } else {
        return PyFloat_FromDouble(BN_NAN);
    }
}

REDUCE_ONE(nanmean, DTYPE0) {
```

```
    npy_DTYPE1 asum;
    INIT_ONE(DTYPE1, DTYPE1)
    BN_BEGIN_ALLOW_THREADS
    if (LENGTH == 0) {
        FILL_Y(BN_NAN)
    } else {
        WHILE {
            asum = 0;
            FOR asum += AI(DTYPE0);
            if (LENGTH > 0) {
                asum /= LENGTH;
            } else {
                asum = BN_NAN;
            }
            YPP = asum;
            NEXT
        }
    }
    BN_END_ALLOW_THREADS
    return y;
}
/* dtype end */
```

So why exactly is Julia great?

Julia is (uniquely) both strongly-typed and dynamic

- *static, compiled, user types* (in C, Fortran, etc)
- *dynamic, interpreted, standard types* (in Python, Mathematica, etc)
- **dynamic, compiled, user types** (Julia)

In dynamic languages the types of variables don't have to be known an runtime (type annotation is optional).

In compiled languages the functions are compiled (properly optimised) and don't suffer from interpretation overhead at runtime.

- Julia's type system has limits, no arrow types (f: $R^n \longrightarrow R^m$)
- The types has less dynamism than in Python (cannot add fields to a type)

So why exactly is Julia great?

Julia has multiple dispatch

> People come to Julia for speed but stay for the multiple dispatch

Difference from OOP (single-dispatch)

```
class Car
    def __init__(self, color, brand):
        # instance variable unique to each instance
        self.color = color
        self.brand = brand

    def information(self):
        return "I'm a {} {}".format(self.color, self.brand)

car = Car("yellow", "mercedes")
car.information()
```

Philosophy: Methods shouldn't belong to a specific data type:

> It makes little sense to artificially deem the operations to "belong" to one argument more than any of the others.

Since there are no classes in Julia, the structures only contain data but no methods

```
struct Car
    color
    brand
end

information(c::Car) = "I'm a $(c.color) $(c.brand)"

car = Car("yellow", "mercedes")
information(car)
```

Levels of dispatch

- none: `f(x1,x2,x3,...)` constant expressive power (e.g., Python functions)
- single: `(x1::T1).f(x2,x3,...)` linear expressive power (e.g., Python class methods)
- multiple `f(x1::T1,x2::T2,x3,...)` exponential expressive power (Julia's core paradigm)

Sharing types

Much simpler to define **new** operations on existing types

- Extending existing operations

```
information(n::Number) = "I'm a number"
```

Give several functions the same name, because they perform conceptually similar tasks, but operate on different types.

- Defining new operations

```
is_beautiful(c::Car) = (c.color == "rose gold")
```

How to do it in OOP? With inheritance one needs to use another name (`NewCar`) or edit the original class or simply drop (single) dispatch.

So why exactly is Julia great?

Julia is garbage-collected

No need to worry about memory management.

Julia supports concurrent, parallel and distributed computing

Natively and with simple and clear syntax.

Julia interfaces particularly well with other languages

Can directly call Python or C.