Zygote

Building a Differentiable Programming Language

mike@juliacomputing.com

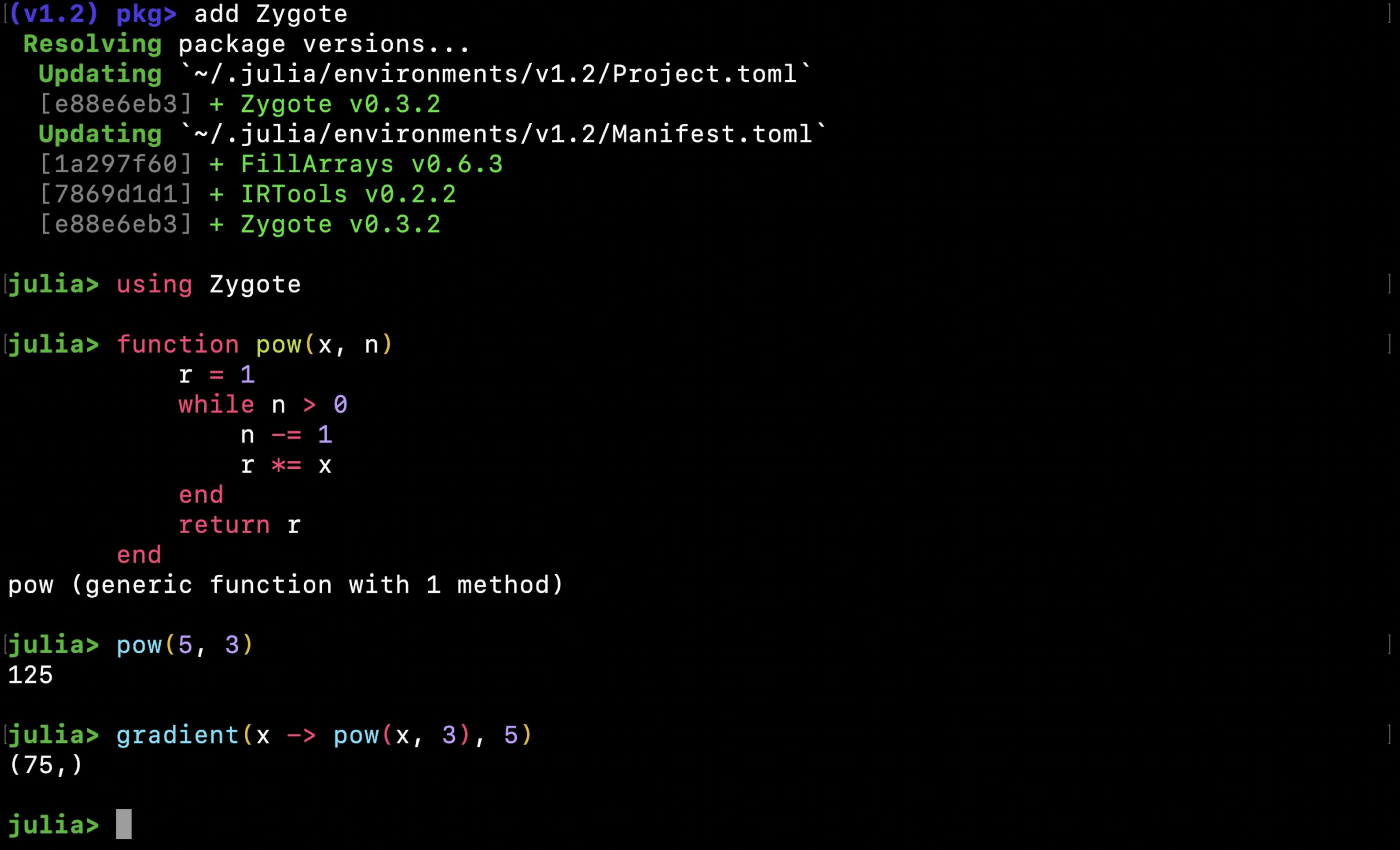
$$y = \sigma(W*x + b)$$

$$y = \sigma(W*x + b)$$

$$\frac{dy}{dx} \longrightarrow$$

How will a small change in the input change the output?

Distance	0.68m
Height	0m
Time	0.03s
Wind Speed	1m/s
Release Angle	45deg


```
(v1.2) pkg> add Zygote
 Resolving package versions...
  Updating `~/.julia/environments/v1.2/Project.toml`
  [e88e6eb3] + Zygote v0.3.2
  Updating `~/.julia/environments/v1.2/Manifest.toml`
  [1a297f60] + FillArrays v0.6.3
  [7869d1d1] + IRTools v0.2.2
  [e88e6eb3] + Zygote v0.3.2

julia> using Zygote

julia> function pow(x, n)
           r = 1
           while n > 0
               n -= 1
               r *= x
           end
           return r
       end
pow (generic function with 1 method)

julia> pow(5, 3)
125

julia> gradient(x -> pow(x, 3), 5)
(75,)

julia> 
```

User Function

```
function pow(x, n)
  r = 1
  while n > 0
    n -= 1
    r *= x
  end
  return r
end
```

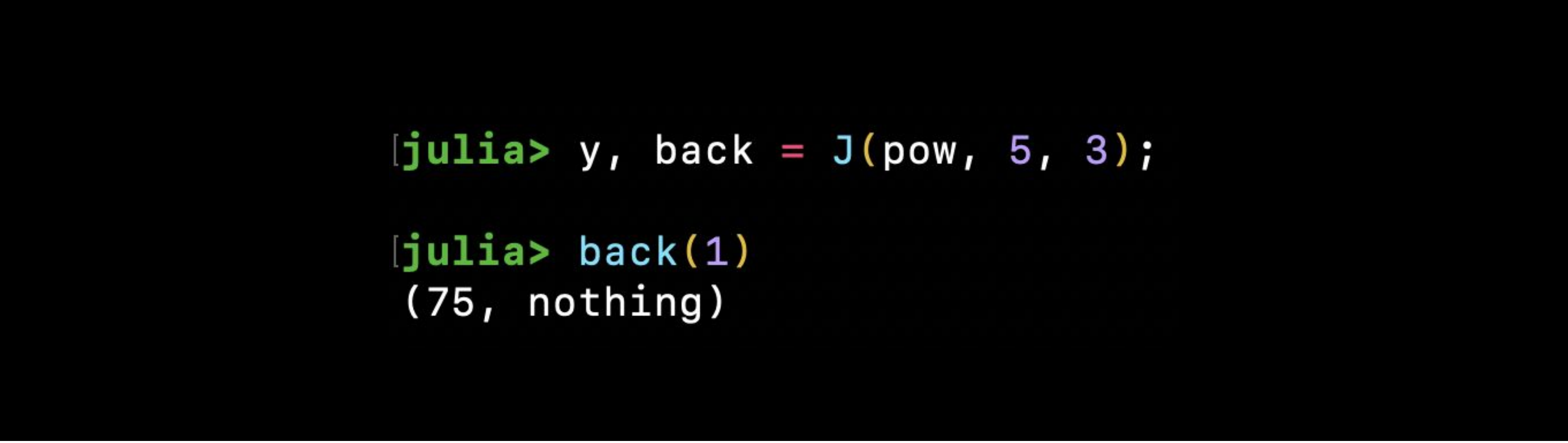
Primal

```
1: (%2, %3)
  br 2 (%3, 1)
2: (%4, %5)
  %6 = %4 > 0
  br 4 unless %6
  br 3
3:
  %7 = %4 - 1
  %8 = %5 * %2
  br 2 (%7, %8)
4:
  return %5
```

Adjoint

```
1: (%1)
  br 2 (%1, 0)
2: (%2, %4)
  br 4 unless @6
  br 3
3:
  %10 = %2 * @2
  %11 = %2 * @5
  %14 = %4 + %11
  br 2 (%10, %14)
4:
  return (%4, 0)
```

```
pow(5, 3) == 125
gradient(pow, 5, 3) == (75, 0)
```



```
julia> y, back = J(pow, 5, 3);

julia> back(1)
(75, nothing)
```

$$y = f(x_1, x_2, \ldots)$$

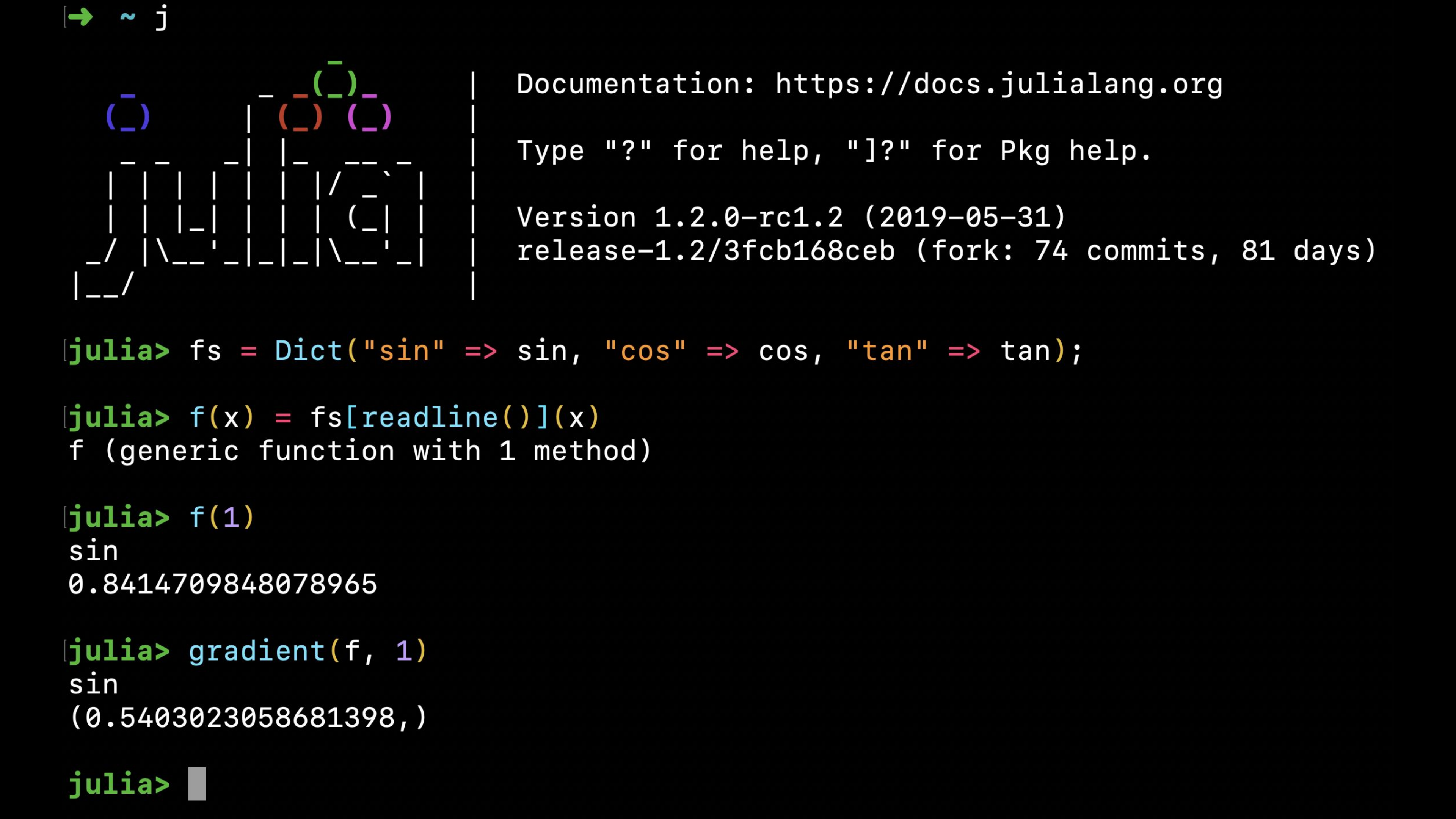
$$y, \mathcal{B} = \mathcal{J}(f, x_1, x_2, \ldots)$$

$$\bar{x}_1, \bar{x}_2, \ldots = \mathcal{B}(\bar{y})$$

```
function foo(x)
  a = bar(x)
  b = baz(a)
  return b
end
```

$\longrightarrow$

```
function J(::typeof(foo), x)
  a, da = J(bar, x)
  b, db = J(baz, a)
  return b, function(b̄)
    ā = db(b̄)
    x̄ = da(ā)
    return x̄
  end
end
```



```
➜  ~ j
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.2.0-rc1.2 (2019-05-31)
 _/ |\__'_|_|_|\__'_|  |  release-1.2/3fcb168ceb (fork: 74 commits, 81 days)
|__/                   |

julia> fs = Dict("sin" => sin, "cos" => cos, "tan" => tan);

julia> f(x) = fs[readline()](x)
f (generic function with 1 method)

julia> f(1)
sin
0.8414709848078965

julia> gradient(f, 1)
sin
(0.5403023058681398,)

julia> 
```

```
J(::typeof(sin), x) = sin(x), ȳ -> ȳ*cos(x)
```

$$\downarrow$$

```
@adjoint sin(x) = sin(x), ȳ -> ȳ*cos(x)
```

Core compiler pass is ~200 lines of code

All semantics added via custom adjoints –
mutation, data structures, checkpointing, etc.

```julia
nestlevel() = 0

@adjoint nestlevel() = nestlevel()+1, _ -> nothing


julia> function f(x)
         println(nestlevel(), " levels of nesting")
         return x
       end

julia> f(1);
0 levels of nesting

julia> grad(f, 1);
1 levels of nesting

julia> grad(x -> x*grad(f, x), 1);
2 levels of nesting
```

```julia
@adjoint hook(f, x) = x, x̄ -> (f(x̄),)

hook(-, x) # reverse the gradient of x

@adjoint checkpoint(f, x...) =
  f(x...), Δ -> J(f, x...)[2](Δ)

@adjoint function forwarddiff(f, x)
  y, J = forward_jacobian(f, x)
  y, Δ -> (J'Δ,)
end
```

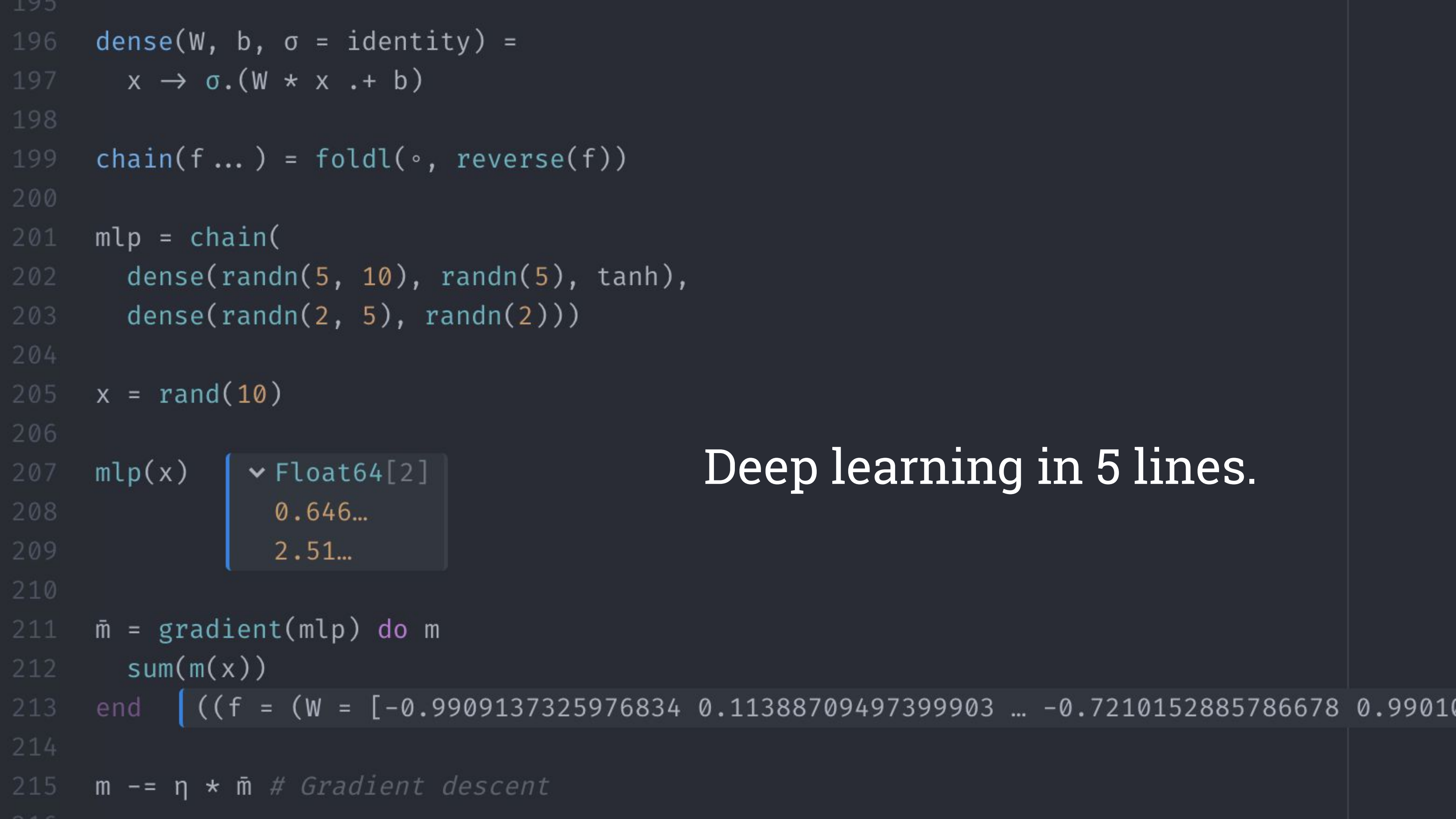
```
julia> hook(f, x) = x
hook (generic function with 1 method)

julia> @adjoint hook(f, x) = x, Δ -> (nothing, f(Δ),)

julia> gradient(2, 3) do a, b
         a*b
       end
(3, 2)

julia> gradient(2, 3) do a, b
         hook(-, a) * b
       end
(-3, 2)

julia> gradient(2, 3) do a, b
         hook(ā -> @show(ā), a) * b
       end
ā = 3
(3, 2)
```

Differentiation á la Carte

- Mixed-mode AD (forward, reverse, Taylor series, …)
- Forward-over-reverse (Hessians)
- Cross-language AD
- Support for Complex and other number types
- Easy custom gradients
- Checkpointing
- Gradient hooks
- Custom types (colours!)
- Hardware backends: CPU, CUDA, TPU, …
- Concurrency, parallelism and distribution
- ~~Deeply nested AD~~ (WIP)

Data Structures & Mutation

```
julia> using Colors

julia> a, b = RGB(1, 0, 0), RGB(0, 1, 0)
(RGB{N0f8}(1.0,0.0,0.0), RGB{N0f8}(0.0,1.0,0.0))

julia> a.r^2
1.0N0f8

julia> gradient(c -> c.r^2, a)
((r = 2.0f0, g = nothing, b = nothing),)

julia> colordiff(a, b)
86.6082357376344

julia> gradient(a -> colordiff(a, b), a)
((r = 0.4590887719632896, g = -9.598786801605689, b = 14.181383399012862),)
```

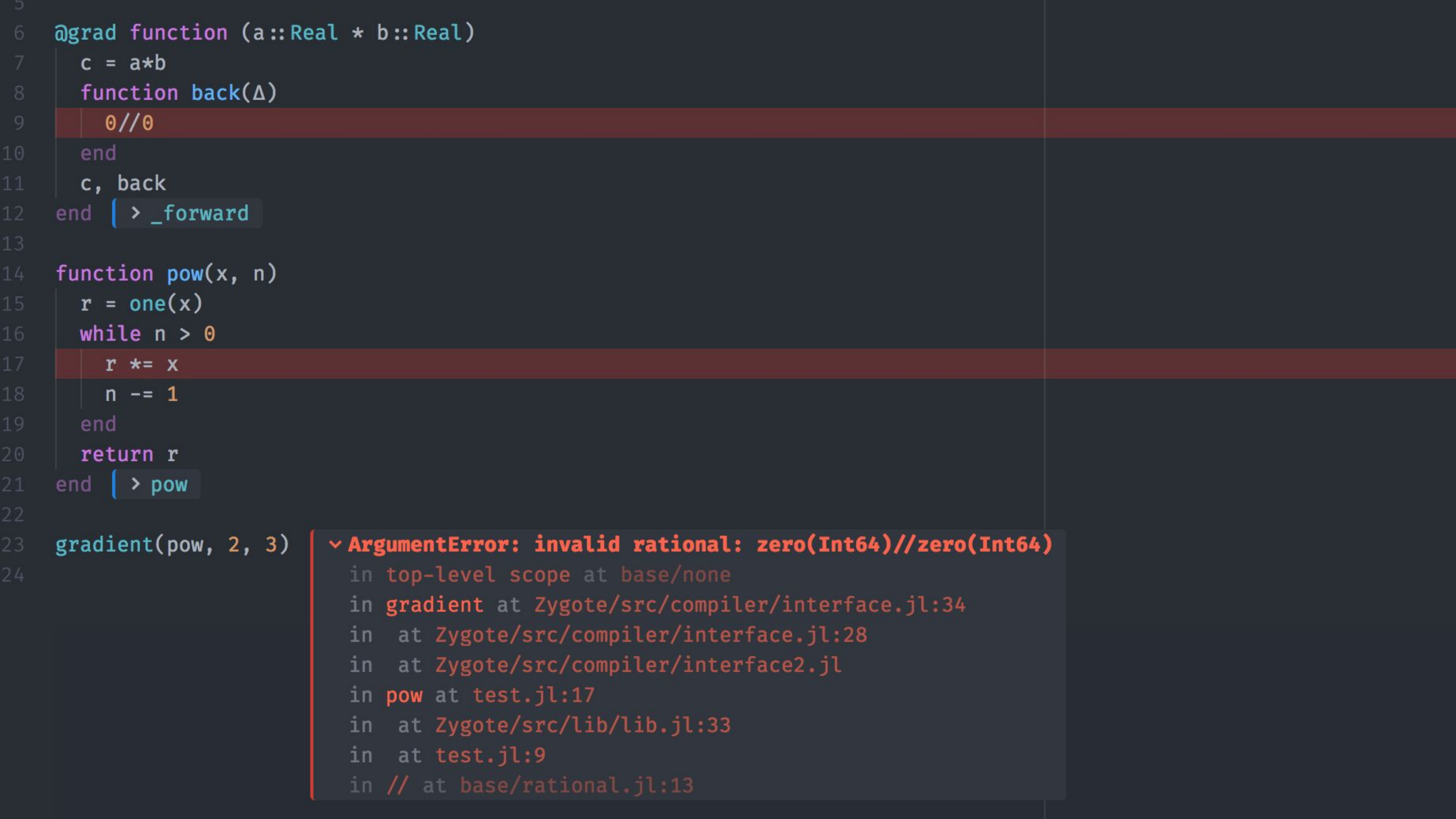


```julia
dense(W, b, σ = identity) =
  x → σ.(W * x .+ b)

chain(f ... ) = foldl(∘, reverse(f))

mlp = chain(
  dense(randn(5, 10), randn(5), tanh),
  dense(randn(2, 5), randn(2)))

x = rand(10)

mlp(x)    ⌄ Float64[2]
            0.646…
            2.51…

m̄ = gradient(mlp) do m
  sum(m(x))
end   ((f = (W = [-0.9909137325976834 0.11388709497399903 … -0.7210152885786678 0.9901

m -= η * m̄ # Gradient descent
```

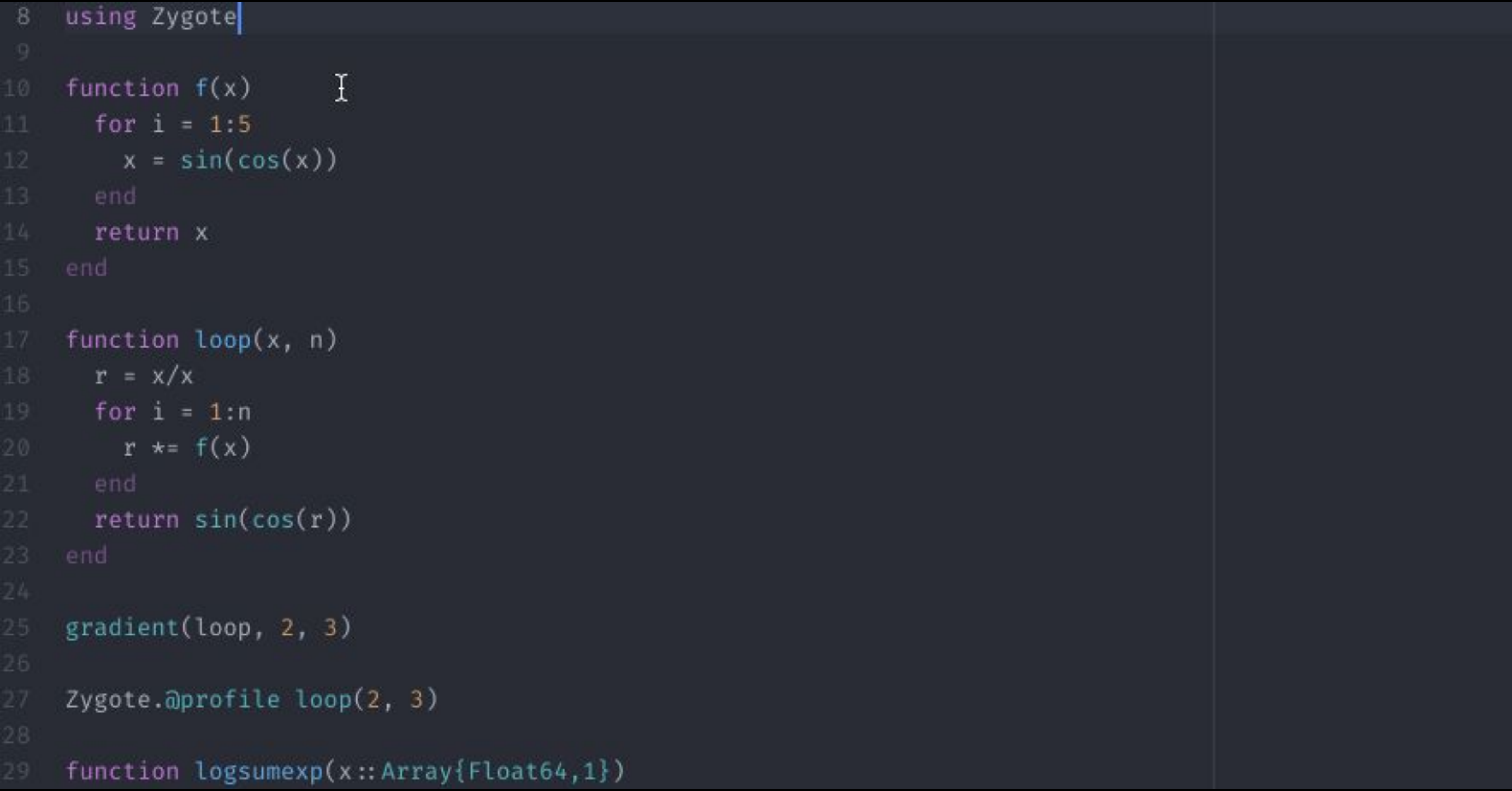
Deep learning in 5 lines.

```
julia> vars = Dict(:r => 0, :n => 0)
Dict{Symbol,Int64} with 2 entries:
  :n => 0
  :r => 0

julia> function pow(x, n)
         vars[:r] = 1
         vars[:n] = n
         while vars[:n] > 0
           vars[:n] -= 1
           vars[:r] *= x
         end
       end
pow (generic function with 1 method)

julia> pow(5, 3); vars[:r]
125

julia> gradient(x -> (pow(x, 3); vars[:r]), 5)
(75,)

julia> vars[:r]
125
```

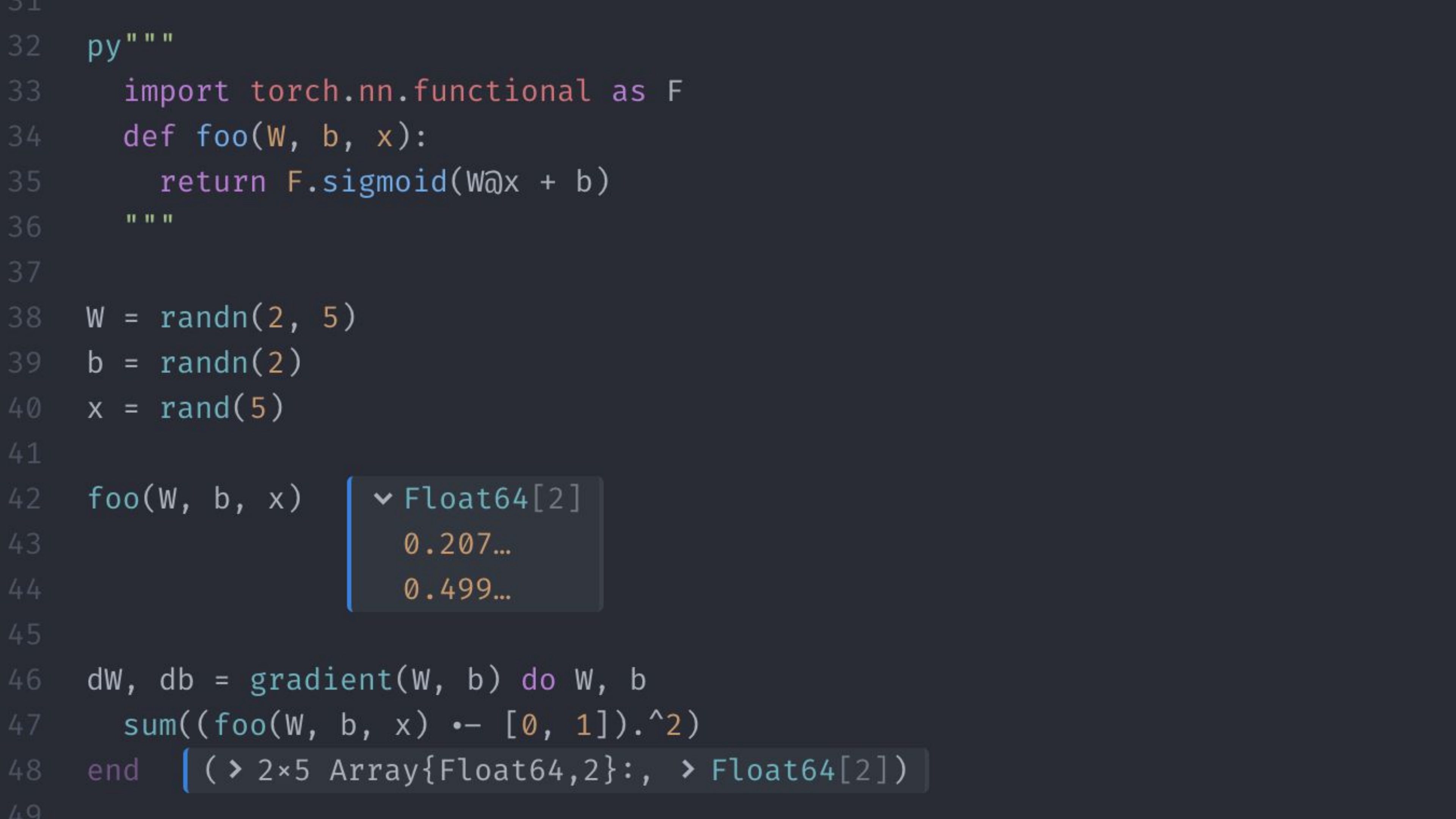
Some Bonus Features


```julia
@grad function (a::Real * b::Real)
  c = a*b
  function back(Δ)
    0//0
  end
  c, back
end  > _forward

function pow(x, n)
  r = one(x)
  while n > 0
    r *= x
    n -= 1
  end
  return r
end  > pow

gradient(pow, 2, 3)
```

```
⌄ ArgumentError: invalid rational: zero(Int64)//zero(Int64)
  in top-level scope at base/none
  in gradient at Zygote/src/compiler/interface.jl:34
  in  at Zygote/src/compiler/interface.jl:28
  in  at Zygote/src/compiler/interface2.jl
  in pow at test.jl:17
  in  at Zygote/src/lib/lib.jl:33
  in  at test.jl:9
  in // at base/rational.jl:13
```



```julia
using Zygote

function f(x)
  for i = 1:5
    x = sin(cos(x))
  end
  return x
end

function loop(x, n)
  r = x/x
  for i = 1:n
    r *= f(x)
  end
  return sin(cos(r))
end

gradient(loop, 2, 3)

Zygote.@profile loop(2, 3)

function logsumexp(x::Array{Float64,1})
```



```
py"""
  import torch.nn.functional as F
  def foo(W, b, x):
    return F.sigmoid(W@x + b)
  """

W = randn(2, 5)
b = randn(2)
x = rand(5)

foo(W, b, x)    ⌄ Float64[2]
                  0.207…
                  0.499…

dW, db = gradient(W, b) do W, b
  sum((foo(W, b, x) .- [0, 1]).^2)
end   (› 2×5 Array{Float64,2}:, › Float64[2])
```

```julia
@adjoint function pycall(f, x...; kw...)
 x = map(py, x)
 y = pycall(f, x...; kw...)
 y.detach().numpy(), function (ȳ)
   y.backward(gradient = py(ȳ))
   (nothing, map(x -> x.grad.numpy(), x)...)
 end
end
```

```julia
function tasks3(x)
  ch = Channel(0)
  @sync begin
    @async put!(ch, x^2)
    take!(ch)
  end
end

@test gradient(tasks3, 5) == (10,)
```

Zygote

Building a Differentiable Programming Language

mike@juliacomputing.com