```
begin
    using Pkg
    Pkg.activate(mktempdir())
    Pkg.add("PyPlot")
    Pkg.add("PlutoUI")
    using PlutoUI,PyPlot
end
```

Number representation

Besides of the concrete names of Julia library functions everything in this chapter is valid for all modern programming languagues and computer systems.

All data in computers are stored as sequences of bits. For concrete number types, the `bitstring` function returns this information as a sequence of `0` and `1`. The `sizeof` function returns the number of bytes in the binary representation.

Integer numbers

```
T_int = Int16
```
```
T_int=Int16
```

```
i = 1
```
```
i=T_int(1)
```

```
2
```
```
sizeof(i)
```

```
"0000000000000001"
```
```
bitstring(i)
```

Positive integer numbers are represented by their representation in the binary system. For negative numbers n, the binary representation of their "two's complement" $2^N - |n|$ (where N is the number of available bits) is stored.

`typemin` and `typemax` return the smallest and largest numbers which can be represented in number type.

```
(-32768, 32767, 32767)
```
```
typemin(T_int),typemax(T_int),2^(8*sizeof(T_int)-1)-1
```

Unless the possible range of the representation $(-2^{N-1}, 2^{N-1})$ is exceeded, addition, multiplication and subtraction of integers are exact. If it is exceeded, operation results wrap around into the opposite sign region.

```
10
```
```
3+7
```

```
-32759
```
```
typemax(T_int)+T_int(10)
```

Floating point numbers

How does this work for floating point numbers ?

```
0.30000000000000004
```

```
0.1+0.2
```

But this should be 0.3. What is happening ???

Real number representation

- Let us think about representation real numbers. Usually we write them as decimal fractions and cut the representation off if the number of digits is infinite.
- Any real number $x \in \mathbb{R}$ can be expressed via the representation formula: $x = \pm \sum_{i=0}^{\infty} d_i \beta^{-i} \beta^e$ with **base** $\beta \in \mathbb{N}, \beta \geq 2$, **significand** (or **mantissa**) digits $d_i \in \mathbb{N}, 0 \leq d_i < \beta$ and **exponent** $e \in \mathbb{Z}$
- The representation is infinite for periodic decimal numbers and irrational numbers.

Scientific notation

The scientific notation of real numbers is derived from this representation in the case of $\beta = 10$. Let e.g. $x = 6.022 \cdot 10^{23}$= `6.022e23` . Then

- $\beta = 10$
- $d = (6, 0, 2, 2, 0 \ldots)$
- $e = 23$

This representation is not unique, e.g. $x_1 = 0.6022 \cdot 10^{24}$= `0.6022e24` $= x$ with

- $\beta = 10$
- $d = (0, 6, 0, 2, 2, 0 \ldots)$
- $e = 24$

IEEE754 standard

This is the actual standard format for storing floating point numbers. It was developed in the 1980ies.

- $\beta = 2$, therefore $d_i \in \{0, 1\}$
- Truncation to fixed finite size: $x = \pm \sum_{i=0}^{t-1} d_i \beta^{-i} \beta^e$
- t : significand (mantissa) length
- Normalization: assume $d_0 = 1 \Rightarrow$ save one bit for the storage of the significand. This requires a normalization step after operations which adjusts significand and exponent of the result.
- k: exponent size. Define L, K: $-\beta^k + 1 = L \leq e \leq U = \beta^k - 1$
- Extra bit for sign
- $\Rightarrow$ storage size: $(t-1) + k + 1$

- Standardized for most modern languages
- Hardware support usually for 64bit and 32bit

precision	Julia	C/C++	k	t	bits
`quadruple`	`n/a`	`long double`	16	113	128
`double`	`Float64`	`double`	11	53	64
`single`	`Float32`	`float`	8	24	32
`half`	`Float16`	`n/a`	5	11	16

- See also the **Julia Documentation on floating point numbers**, **0.30000000000000004.com**, **wikipedia** and the links therein.

The storage sequence is: Sign bit, exponent, mantissa.

Storage layout for a normalized Float32 number ($d_0 = 1$):

- bit 1: sign, $0 \rightarrow +, \quad 1 \rightarrow -$
- bit $2 \ldots 9$: $k = 8$ exponent bits
  - the value $e + 2^{k-1} - 1 = e + 127$ is stored $\Rightarrow$ no need for sign bit in exponent
- bit $10 \ldots 32$: $23 = t - 1$ mantissa bits $d_1 \ldots d_{23}$
- $d_0 = 1$ not stored $\equiv$ "hidden bit"

Julia allows to obtain the signifcand and the exponent of a floating point number

```
x0 = 2.0
x0=2.0
```

```
(1.0, 1)
significand(x0),exponent(x0)
```

- We can calculate the length of the exponent k from the maximum representable floating point number by taking the base-2 logarithm of its exponent:

```
exponent_length(T::Type{<:AbstractFloat})=Int(log2(exponent(floatmax(T))+1)+1);
```

- The size of the significand t is calculated from the overall size of the representation minus the size of the exponent and the size of the sign bit + 1 for the "hidden bit".

```
significand_length(T::Type{<:AbstractFloat})=8*sizeof(T)-exponent_length(T)-1+1;
```

This allows to define a more readable variant of the bitstring repredentatio for floats.

- The sign bit is the first bit in the representation:

```
signbit(x::AbstractFloat)=bitstring(x)[1:1];
```

- Next comes the exponent:

```
exponent_bits (generic function with 1 method)
exponent_bits(x::AbstractFloat)=bitstring(x)[2:exponent_length(typeof(x))+1]
```

- And finally, the significand:

```
significand_bits(x::AbstractFloat)=bitstring(x)
[exponent_length(typeof(x))+2:8*sizeof(x)];
```

- Put them together:

```
floatbits(x::AbstractFloat)=signbit(x)*"_"*exponent_bits(x)*"_"*significand_bits(x);
```

Julia floating point types

```
T = Float16
T=Float16
```

Type Float16:

- size of exponent: 5
- size of significand: 11

```
x = Float16(0.1)
x=T(0.1)
```

- Binary representation: 0_01011_1001100110
- Exponent e=-4
- Stored: e+15= 11
- $d_0 = 1$ assumed implicitely.

- Numbers which are exactly represented in decimal system may not be exactly represented in binary system!
- Such numbers are always rounded to a finite approximate

```
x_per = Float16(0.2998)
```
```
x_per=T(0.1)+T(0.2)
```

```
"0_01101_0011001100"
```
```
floatbits(x_per)
```

Floating point limits

- Finite size of representation $\Rightarrow$ there are minimal and maximal possible numbers which can be represented
- symmetry wrt. 0 because of sign bit
- smallest positive denormalized number: $d_i = 0, i = 0 \dots t-2, d_{t-1} = 1 \Rightarrow x_{min} = 2^{1-t}2^L$

```
(6.0e-8, "0_00000_0000000001")
```
```
nextfloat(zero(T)), floatbits(nextfloat(zero(T)))
```

- smallest positive normalized number: $d_0 = 1, d_i = 0, i = 1 \dots t-1 \Rightarrow x_{min} = 2^L$

```
(6.104e-5, "0_00001_0000000000")
```
```
floatmin(T),floatbits(floatmin(T))
```

- largest positive normalized number: $d_i = 1, 0 \dots t-1 \Rightarrow x_{max} = 2(1-2^{1-t})2^U$

```
(6.55e4, "0_11110_1111111111")
```
```
floatmax(T), floatbits(floatmax(T))
```

- Largest representable number:

```
(Inf, "0_11111_0000000000", 6.55e4, "0_11110_1111111111")
```
```
typemax(T),floatbits(typemax(T)),prevfloat(typemax(T)),
floatbits(prevfloat(typemax(T)))
```

Machine precision

- There cannot be more than 2^{t+k} floating point numbers $\Rightarrow$ almost all real numbers have to be approximated
- Let x be an exact value and $\tilde{x}$ be its approximation. Then $|\frac{\tilde{x}-x}{x}| < \epsilon$ is the best accuracy estimate we can get, where
  - $\epsilon = 2^{1-t}$ (truncation)
  - $\epsilon = \frac{1}{2}2^{1-t}$ (rounding)
- Also: ϵ is the smallest representable number such that $1 + \epsilon > 1$.
- Relative errors show up in particular when
  - subtracting two close numbers
  - adding smaller numbers to larger ones

How do operations work?

E.g. Addition

- Adjust exponent of number to be added:
  - Until both exponents are equal, add 1 to exponent, shift mantissa to right bit by bit
- Add both numbers
- Normalize result

The smallest number one can add to 1 can have at most t bit shifts of normalized mantissa until mantissa becomes 0, so its value must be 2^{-t}.

Machine epsilon

- Smallest floating point number ϵ such that $1+\epsilon>1$ in floating point arithmetic
- In exact math it is true that from $1+\varepsilon=1$ it follows that $0+\varepsilon=0$ and vice versa. In floating point computations this is not true

```
ϵ = Float16(0.000977)
```
```julia
ϵ=eps(T)
```

```
"0_00101_0000000000"
```
```julia
floatbits(ϵ)
```

```
(1.0, "0_01111_0000000000", "0_01111_0000000000")
```
```julia
one(T)+ϵ/2,floatbits(one(T)+ϵ/2), floatbits(one(T))
```

```
(1.001, "0_01111_0000000001")
```
```julia
 one(T)+ϵ,floatbits(one(T)+ϵ)
```

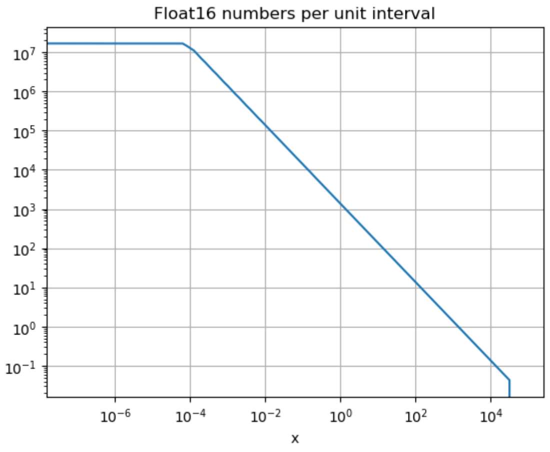
```
(0.000977, "0_00101_0000000000")
```
```julia
nextfloat(one(T))-one(T),floatbits(nextfloat(one(T))-one(T))
```

Density of floating point numbers

How dense are floating point numbers on the real axis?

```julia
function fpdens(x::AbstractFloat;sample_size=1000)
    xleft=x
    xright=x
    for i=1:sample_size
        xleft=prevfloat(xleft)
        xright=nextfloat(xright)
    end
    return prevfloat(2.0*sample_size/(xright-xleft))
end;
```

```
X =
 Float16[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0
```
```julia
X=T(10.0) .^collect(-10:T(0.1):10)
```



```
begin
    fig=PyPlot.figure()
    PyPlot.loglog(X,fpdens.(X))
    PyPlot.title("$(eltype(X)) numbers per unit interval")
    PyPlot.grid()
    PyPlot.xlabel("x")
    fig
end
```