Julia 1: Identifiers, simple types, basic operations

Bernt Lie

University of South-Eastern Norway

May, 2021; minor update September 2023

Learning goals

In this session, we will look at:

- valid Julia identifiers + naming conventions
- give an overview of simple types
- give an overview of basic functions

Julia identifiers

What is an identifier?

By *identifier* (label) is meant a symbolic name for items/memory cells that are introduced by Julia or the user. Some general conventions in computer science:

- **CamelCase** (= **CapitalizedWord** = **CapWords**) where each word starts with a capital letter
- **mixedCase** where the first word starts with a lower case letter while each subsequent word starts with a capital letter
- **underscore_separated_words** where each word is separated by underscore
- **combinedwords** where words are simply concatenated

Valid symbols in identifiers

Variable names must begin with a letter (A-Z or a-z), underscore ("_"), or a subset of Unicode code points greater than 00A0. [A code point is a number assigned to represent an abstract character in a system for representing text (such as Unicode). In Unicode, a code point is expressed in the form "U+1234" where "1234" is the assigned number. For example, the character "A" is assigned a code point of U+0041].

Subsequent characters may also include ! and digits (0-9 and some other characters), as well as other Unicode code points: diacritics and other modifying marks (categories Mn/Mc/Me/Sk), some punctuation connectors (category Pc), primes, and a few other characters.

Operators such as + are also valid identifiers, but are parsed specially. In some contexts, operators can be used just like variables; for example (+) refers to the addition function, and (+) = f will reassign it.

Assignment is expressed by symbol `=` in Julia. Thus, e.g., `x = 2+3` means that the value (here: `5`) on the right-hand-side of symbol `=` is *assigned* to identifier `x` on the left-hand-side of symbol `=` ; in other words "take the evaluated expression `2+3 -> 5` on the right-hand-side of symbol `=` and put this value into the memory location given by identifier `x` ".

The only explicitly *illegal* names for variables are the names of the built-in Keywords, which at Julia v.1.9.3 are:

```
PS C:\Users\Bernt_Lie> julia --lisp
;  _
; |_ _ _ |_ _ |  . _ _
; | (-||||_(_)|__|_)|_)
;-------------------|----------------------------------------------------------
-

> reserved-words
(begin while if for try return break continue function macro quote let local
 global const do struct module baremodule using import export end else elseif
 catch finally true false)
```

Examples:

In [532…

```
x = 5.0
y = 3+2im      # = 3+2*j = 3+2*i, complex number
α = 3.5        # \alpha + TAB gives α; use LaTeX syntax
Γ₁ = -2        # \Gamma + TAB followed by \_1 + TAB
⏰ = 8.00      # \: + TAB gives intellisence pop-up menu
;              # semicolon suppresses output
```

In [533…

```
pi
```

```
π = 3.1415926535897...
```

Observe that `pi` is the mathematical constant π, which can also be written as Unicode character \pi + TAB.

Using non-ASCII symbols should be used with care, as some word processors don't support it.

Identifier conventions in Julia

- **Label** ("variable"): Lower case, word separation by underscore if a combined words sequence is difficult to read. *Examples*: `temperature` , `temperature_indoor` , `Tin` , `T_in`
- **Types and Modules**: *CamelCase* (not *mixedCase*). *Examples*: `Float64` , `Base` , `DifferentialEquations`
- **Functions and macros**: Lower case + avoid underscore. If function combines several actions, underscore may be needed; alternatively: split up into several functions. *Examples*: `println` , `isequal` , `timed` . *Note*: macros are specified with symbol `@` at the start, e.g., macro `benchmark` is invoked by `@benchmark` .

- **Mutating functions**: Functions that modify their arguments *by convention* ends with exclamation mark `!` (sometimes read out loud as "bang"). *Examples*: `plot` is non-mutating (creates a fresh plot), `plot!` ("plot bang") is mutating (modifies an existing plot).

Unevaluated identifiers

It is possible to tell Julia that the expression on the right-hand-side of an assignment should be assigned to the identifier *without evaluating the right-hand-side* -- this is done by prepending the identifier with `:`. In other words: `:x=2+3` means "assign the unevaluated expression `2+3` to identifier `x`. Unevaluated assignment is used in more advanced methods involving so-called *macros*, and is not discussed much in this introduction.

Memory location of identifiers

Julia does not provide direct information about the location in computer memory of an identifier. However, utility function `objectid(x)` provides a hexademical "expression" of the location, which means that two objects whose identifer has the same `objectid`, will have the same memory location, and thus be identical.

Note also that the memory location of an identifier is really a pointer to the first byte of the stored object. Objects taking more space than one byte may be spread over non-contiguous memory space; the data structure that is stored at the `objectid` location will hold information about where the object is stored in memory, and the user does not need to keep track of this.

However, the `objectid` function gives information about whether identifiers are moved in memory, etc.

Comments and multiline statement continuation

Anything on a line after character `#` is considered a comment, and is not executed.

Multiline comments are enclosed in `#=` and `=#`.

Any incomplete statement at line break leads to an assumed continuation of the statement to the next line.

In [534…

```
# This is a comment
a = 3 # This is also a comment
```

3

In [535…

```
#=
We can have multiline
comments, too
=#
a = 3
```

3

In [536…

```
# Because each of the lines *at line break* are complete statements, the
# following code is treated as to separate statements
a = 3
+ 3
```

3

In [537…

```
# Because the statement of the first line is incomplete *at line break*,
# the statement is assumed to continue on the next line
a = 3 +
3
```

6

Simple types

Checking type

Julia comes with built-in simple types (abstract and concrete types), collections, and the possibility of user-defined types.

Type information can be queried by base function `typeof()` where the argument is the type or an identifier holding the type.

As we will see later, the `typeof()` function also returns information about collection structure. For collections, we can query the *element type* by function `eltype()` . For scalars (as here), the result of `typeof()` and the result of `eltype()` will be the same.

Memory requirement (in Bytes) can be queried by `sizeof()` where the argument is the identifier.

In [538…

```
typeof(x)
```

```
Float64
```

In [539…

```
typeof(y)
```

```
Complex{Int64}
```

In [540…

```
typeof(⏰)
```

```
Float64
```

In [541…

```
sizeof(x)
```

8

In [542…

```
sizeof(⏰)
```

8

In [543…

```
sizeof(y)
```

16

Character

A *character* type `Char` is given by a single symbol enclosed in single quotation marks, e.g., `'c'`, `'⏰'`. NOTE: a character is not the same as a string.

In [544…

```
x = 'a'
```

'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

In [545…

```
typeof(x)
```

Char

In [546…

```
y = '⏰'
```

'⏰': Unicode U+23F0 (category So: Symbol, other)

In [547…

```
typeof(y)
```

Char

Number

Types and subtypes

A (super-) type is related to a subtype as `subtype <: supertype`, where symbol `<:` means "is subtype of", and in some ways is similar to a subset. Essentially, if some code works for a supertype, it should work for its subtypes.

Abstract number types

- `Number` is an abstract supertype for all number types
- `Real` is a supertype for all real numbers, and is a subtype of `Number` (`Real <: Number` in Julia syntax, where symbol `<:` means subtype and is in some ways similar to subset)
- `AbstractFloat` is a supertype for all floating point numbers; `AbstractFloat <: Real`.
- `AbstractIrrational` represents an exact irrational value; `AbstractIrrational <: Real`. In computations, an `AbstractIrrational` is rounded to correct precision upon operations with other types.
- `Integer` is a supertype of all integer numbers; `Integer <: Real`.
- `Signed` is a supertype of all signed integer numbers; `Signed <: Integer`.
- `Unsigned` is a supertype of all unsigned integer numbers; `Unsigned <: Integer`.

Concrete number types

- `Float16` is an IEEE 16 bit floating point number; `Float16 <: AbstractFloat`

- `Float32` is an IEEE 32 bit floating point number; `Float32 <: AbstractFloat`
- `Float64` is an IeEE 64 bit floating point number; `Float64 <: AbstractFloat`
- `BigFloat` is a variable precision floating point number; `BigFloat <: AbstractFloat`
- `Bool` is a boolean number (value: `true` or `false`); `Bool <: Integer` . *Note* : `false` is numerically equal to `0` and `true` is numerically equal to `1`
- `Int8` is a signed 8 bit integer; `Int8 <: Signed`
- `Int16` is a signed 16 bit integer; `Int16 <: Signed`
- `Int32` is a signed 32 bit integer; `Int32 <: Signed`
- `Int64` is a signed 64 bit integer; `Int64 <: Signed`
- `Int128` is a signed 128 bit integer; `Int128 <: Signed`
- `BigInt` is a signed variable precision integer; `BigInt <: Signed`
- `UInt8` is an unsigned 8 bit integer; `UInt8 <: Unsigned`
- `UInt16` is an unsigned 16 bit integer; `UInt16 <: Unsigned`
- `UInt32` is an unsigned 32 bit integer; `UInt32 <: Unsigned`
- `UInt64` is an unsigned 64 bit integer; `UInt64 <: Unsigned`
- `UInt128` is an unsigned 128 bit integer; `UInt128 <: Unsigned`
- `Complex` is a complex number; `Complex <: Number` . The real and imaginary numbers can be of various types.
- `Rational` is a rational number; `Rational <: Real` . The numerator and denominators are of type `Integer`
- `Irrational` is an irrational number; `Irrational <: AbstractIrrational` .

Creating numbers

- By default, a consecutive sequence of numbers without decimal point becomes an integer word size matching the Operating System/Julia version. Thus, `123` is of type `Int64` by default on a 64 bit computer. System label `Sys.WORD_SIZE` holds the machine word size.

- By default, a consecutive sequence of numbers with one decimal point becomes a floating point number matching the OS/Julia word size, hence `1.23` is of type `Float64` by default on a 64 bit computer.

- In order to increase readability, underscore `_` can be inserted to group digits, e.g., `12_103` is easier to read than `12103` ; both numbers have the same value in Julia.

- To specify a certain type for a number, the (Absctract or Concrete) type name operates as an instantiator.

- If possible a number can be converted from the existing type to a new type using the type name as a convertion function.

- Negative numbers (except unsigned numbers) are created by prepending the number with symbol `-` . Prepending *Unsigned* numbers with `-` creates the complement number.

Some integers

In [548…

```
Sys.WORD_SIZE
```

64

In [549…

```
x = 123
```

123

In [550…

```
typeof(x)
```

Int64

In [551…

```
objectid(x)     # Observe the hex number for the object identification
```

0x1cf1facf3ea7e006

In [552…

```
x = Int8(123)
```

123

In [553…

```
typeof(x)
```

Int8

In [554…

```
objectid(x)     # Observe that by re-assigning a new value to `x`, the new
                # value is stored in a different memory location than
                # above
```

0x0000000027a694ee

In [555…

```
x = 1_2_3
```

123

In [556…

```
typeof(x)
```

Int64

In [557…

```
x = BigInt(123)
```

123

In [558…

```
typeof(x)
```

BigInt

In [559…

```
x = false
```

false

In [560…

```
typeof(x)
```

```
Bool
```

In [561…
```julia
x = Bool(1)
```
```
true
```

In [562…
```julia
typeof(x)
```
```
Bool
```

In [563…
```julia
x = -1
```
```
-1
```

In [564…
```julia
typeof(x)
```
```
Int64
```

In [565…
```julia
x = -UInt64(1)
```
```
0xffffffffffffffff
```

In [566…
```julia
typeof(x)
```
```
UInt64
```

Hexadecimal, octal, binary integers. Unsigned integers

Hexidecimal integers are written with prefix `0x` followed by hexadecimal digits `0,...,9,a,...,f` . Octale integers are input with prefix `0o` followed by octal digits `0,...,7` , but are internally represented as hexadecimal numbers. Binary integers are input with prefix `0b` followed by binary digits `0,1` , but are internally represented as hexadecimal numbers.

Unsigned integers may be input as hexadecimal numbers.

In [567…
```julia
x = 0x1b
```
```
0x1b
```

In [568…
```julia
Int(x)
```
```
27
```

In [569…
```julia
x = 0o17
```
```
0x0f
```

In [570…
```julia
Int(x)
```
```
15
```

In [571...

```
x = 0b10011
```

```
0x13
```

In [572...

```
Int(x)
```

```
19
```

In [573...

```
UInt8(19)
```

```
0x13
```

Some real numbers

In [574...

```
x = 1.23
```

```
1.23
```

In [575...

```
typeof(x)
```

```
Float64
```

In [576...

```
x = Float32(1.23)
```

```
1.23f0
```

In [577...

```
x = BigFloat(1.23)
```

```
1.229999999999999982236431605997495353221893310546875
```

In [578...

```
typeof(x)
```

```
BigFloat
```

In [579...

```
x = 12/13
```

```
0.9230769230769231
```

In [580...

```
typeof(x)
```

```
Float64
```

Hexadecimal floating point numbers

Hexadecimal floating point numbers can be written with notation `xpn` where `x` is a hexadecimal mantissa and `n` is an decimal integer; such numbers can be represented as `Float64` .

In [581...

```
x = 0xb.acap-2
```

```
2.9185791015625
```

In [582…

```
typeof(x)
```

Float64

Some rational numbers

In [583…

```
x = 12//13
```

12//13

In [584…

```
typeof(x)
```

Rational{Int64}

In [585…

```
x = Int16(12)//Int16(13)
```

12//13

In [586…

```
typeof(x)
```

Rational{Int16}

In [587…

```
x = Int16(12)//Int32(13)
```

12//13

In [588…

```
typeof(x)
```

Rational{Int32}

In [589…

```
x = Rational(12,13)
```

12//13

Some complex numbers

In [590…

```
x = 12+3im
```

12 + 3im

In [591…

```
typeof(x)
```

Complex{Int64}

In [592…

```
x = Complex(12,3)
```

12 + 3im

Some number conversion

In [593…
```
x = 123
```
```
123
```
In [594…
```
typeof(x)
```
```
Int64
```
In [595…
```
x = Int32(x)
```
```
123
```
In [596…
```
typeof(x)
```
```
Int32
```
In [597…
```
x = BigFloat(1.23)
```
```
1.229999999999999982236431605997495353221893310546875
```
In [598…
```
Float64(22//7)  # A simple rational accproximation of π
```
```
3.142857142857143
```
In [599…
```
x = Float32(x)
```
```
1.23f0
```
`x = Int64(x)` # this is not possible: it is necessary to first round a floating point number to a whole number before it can be converted to an integer; see section on functions.

Floating point numbers in form $x \cdot 10^n$

A number $x \cdot 10^n$ where x is an integer or floating point number and n is an integer, is written as `xen` (`Float64`) or `xfn` (`Float32`), e.g., $1.0 \cdot 10^{-2}$ is written `1.0e-2` , which leads to a `Float64` type, or as `1.0f-2` which leads to a `Float32` type.

NOTE: do *not* write $x \cdot 10^n$ as `x*10^n` ; see section on *Algebraic operations* below and the warning about *type overflow*.

In [600…
```
x = -3.2e2
```
```
-320.0
```
In [601…
```
typeof(x)
```
```
Float64
```
In [602…
```
x = Float32(x)
```

```
-320.0f0
```

In [603…

```
x = -3.2f2
```

```
-320.0f0
```

Number properties and special numbers

The largest and smallest numbers that can be represented by a type is given by functions `typemax()` and `typemin()` , where the argument is a type or a number with given type.

The machine precision for *floating point* numbers is the distance between `1.0` in the given type, and the smallest larger number which can be distinguised. The machine precision indicates the possible accuracy with the given type, and is found by function `eps()` (epsilon) where the argument is a type or a number with given type.

For *floating point numbers*, symbols for (plus/minus) infinity as well as not being a number like any other number (including itself), exists.

- **Positive infinity**: `Inf` (`Float64`), `Inf32` (`Float32`), `Inf16` (`Float16`)
- **Negative infinity**: `-Inf` (`Float64`), `-Inf32` (`Float32`), `-Inf16` (`Float16`)
- **Not-a-number**: `NaN` (`Float64`), `NaN32` (`Float32`), `NaN16` (`Float16`)

In [604…

```
# max/min type values
typemax(Int16)
```

```
32767
```

In [605…

```
typemin(Int16)
```

```
-32768
```

In [606…

```
typemax(UInt16)
```

```
0xffff
```

In [607…

```
Int32(typemax(UInt16))
```

```
65535
```

In [608…

```
typemin(UInt16)
```

```
0x0000
```

In [609…

```
# Floating point machine precision
eps(Float64)
```

```
2.220446049250313e-16
```

In [610…

```
x = 1.7
```

```
1.7
```

In [611…

```
eps(x)
```

```
2.220446049250313e-16
```

In [612…

```
eps(Float32)
```

```
1.1920929f-7
```

In [613…

```
eps(BigFloat)
```

```
1.727233711018888925077270372560079914223200072887256277004740694033718360632485e-77
```

In [614…

```
# Special floating point numbers
1/Inf
```

```
0.0
```

In [615…

```
1/(-Inf)
```

```
-0.0
```

In [616…

```
Inf/Inf
```

```
NaN
```

In [617…

```
Inf-Inf
```

```
NaN
```

Mathematical operations

The following is a simplified overview.

Algebraic operations

Infix notation for addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), power (`^`); grouping of operation by parenthesis in the normal way. [*Infix* implies that a binary operator (say) `+` is placed *between* the two arguments such as in `a+b` ; the alternative is a function notation `+(a,b)` .]

- For *multiplication*, the multiplication operator (`*`) can be *skipped* between an identifier and a number (just as in mathematics), e.g., with identifier `x` , `3x` is the same as `3*x`
- If the operations are applied to different datatypes, then the numbers are automatically converted to the most general datatype before execution.
- Division (`/`) leads to a floating point number.
- If division between integers is meant to denote integer division, use operator `//` , which leads to an exact rational number.

- Taking the power between two integers always leads to an integer. **Note**: beware of the possibility of type overflow

In [618…
```
1+3
```
```
4
```

In [619…
```
1. + 3
```
```
4.0
```

In [620…
```
3-4
```
```
-1
```

In [621…
```
3*4.
```
```
12.0
```

In [622…
```
12/3
```
```
4.0
```

In [623…
```
12//3
```
```
4//1
```

In [624…
```
3^4.
```
```
81.0
```

In [625…
```
3.0^4
```
```
81.0
```

In [626…
```
3^4
```
```
81
```

In [627…
```
10^18
```
```
1000000000000000000
```

In [628…
```
10^19
```
```
-8446744073709551616
```

In [629…
```
# Observe that 10^19 is an integer which is larger than typemax(Int), which
# leads to the complement number
Float64(typemax(Int))
```

```
9.223372036854776e18
```

In [630…

```
# To avoid overflow, use exponent notation!, which produces a
# floating point number.
1e19
```

```
1.0e19
```

In [631…

```
1+2/(3-4)
```

```
-1.0
```

Updating operators

- **Addition**: `x = x + y` can be written as `x += y`
- **Subtraction**: `x = x - y` can be written as `x -= y`
- **Multiplication**: `x = x*y` can be written `x *= y`
- **Division**: `x = x/y` can be written as `x /= y` ; `x = x\y` can be written as `x \= y` ; `x = x÷y` can be written as `x ÷= y`
- **Power**: `x = x^y` can be written as `x ^= y`

In [632…

```
x = 3
x = x^2
```

```
9
```

In [633…

```
x = 3
x ^= 2
```

```
9
```

Relational operators

Relational operators are operators on various datatypes which gives a Boolean outcome. Infix notation for equality (`==`), greater than (`>`), greater or equal than (`>=`), less than (`<`), less than or equal (`<=`), not equal to (`!=`), identical equal to (`===`), not identical equal to (`!==`); grouping of operation by parenthesis in the normal way.

Because floating point numbers are approximated in a binary representation, we really often will accept numbers as equal if they are *approximatly* equal. Julia privides function `isapproximate()` for that.

In [634…

```
Float64(1/2) == Float32(1/2)    # 0.5 = 2^(-1)  can be represented exactly
                                # in binary number systems
```

```
true
```

In [635…

```
Float64(1/3) == Float32(1/3)    # 1/3 can *not* be represented exactly
                                # in binary number systems
```

```
false
```

In [636…

```
Float64(1/3) > Float32(1/3)
```

```
false
```

In [637…

```
Float64(1/3) < Float32(1/3)
```

```
true
```

In [638…

```
pi ≤ 3.15    # Using LaTeX notation, \le + TAB gives Unicode symbol ≤
```

```
true
```

In [639…

```
Float64(1/3) != Float32(1/3)
```

```
true
```

In [640…

```
x = 2
y = x
z = 2
```

```
2
```

In [641…

```
x === y
```

```
true
```

In [642…

```
x === z
```

```
true
```

In [643…

```
x !== z
```

```
false
```

In [644…

```
<(2,3)
```

```
true
```

In [645…

```
isapprox(Float64(1/3),Float32(1/3))
```

```
true
```

Logical operators

Logical operators are operators on the `Bool` datatype which gives a Boolean outcome. Infix notation for *negation* (`!`), logical *and* (`&&`), logical *or* (`||`); grouping of operation by parenthesis in the normal way.

Observe that logical *and* and logical *or* are *short-circuiting* in the sense that in `a && b`, `b` is not evaluated if `a` is `false`, while in `a || b`, `b` is not evaluated if `a` is `true`. This short-circuiting helps speed up the execution of the code.

In [646…

```
true == false
```

```
false
```

In [647…

```
(2<3) && (5>4)
```

```
true
```

In [648…

```
(2<3) || (5<4)
```

```
true
```

Mathematical functions

Algebraic operations as functions

Infix operators can also be written as functions:

- **Addition**: `x+y+...+z` can be written as `+(x,y,...,z)`
- **Subtraction**: `x-y` can be written as `-(x,y)`
- **Multiplication**: `x*y*...*z` can be written as `*(x,y,...,z)`
- **Division**: `x/y` can be written as `/(x,y)`
- **Left division**: `x\y` can be written as `\(x,y)`
- **Power**: `x^y` can be written as `^(x,y)`

In [649…

```
2+5, +(2,5)
```

```
(7, 7)
```

In [650…

```
2/3, /(2,3)
```

```
(0.6666666666666666, 0.6666666666666666)
```

In [651…

```
2^3, ^(2,3)
```

```
(8, 8)
```

In [652…

```
2*3*4, *(2,3,4)
```

```
(24, 24)
```

Relational operators as functions

Infix relational operators can be writtens as functions:

- **Equality**: `x==y` can be written as `==(x,y)`; `x!=y` can be written as `!=(x,y)`; `x===y` can be written as `===(x,y)`; `x!==y` can be written as `!==(x,y)`.
- **Larger than**: `x>y` can be written as `>(x,y)`; `x>=y` can be written as `>=(x,y)`.
- **Less than**: `x>y` can be written as `<(x,y)`; `x<=y` can be written as `<:(x,y)`.

Because computers use finite wordlength binary representation, floating point numbers can not be represented exactly. We are therefore often interested in whether numbers are *approximately* equal, within the given machine precision. Julia provides function `isapprox(x,y)` for that.

In [653…

```
==(Float64(1/3),Float32(1/3))
```

```
false
```

In [654…

```
isapprox(Float64(1/3),Float32(1/3))
```

```
true
```

Basic mathematical functions: single argument

The following *single argument* mathematical functions are given in the on-line Julia documentation at https://julialang.org/:

- **Trigonometric [radians]**: `sin`, `cos`, `tan`, `sec`, `csc`, `cot` represent $\sin(x)$, $\cos(x)$, $\tan(x)$, $\sec(x) = \frac{1}{\cos(x)}$, $\csc(x) = \frac{1}{\sin(x)}$, $\cot(x) = \frac{1}{\tan(x)}$, respectively, where x is given in $\mathrm{radians}$.

- **Trigonometric [degrees]**: `sind`, `cosd`, `tand`, `secd`, `cscd`, `cotd` represent $\sin(x)$, $\cos(x)$, $\tan(x)$, $\sec(x)$, $\csc(x)$, $\cot(x)$, respectively, where x is given in $\mathrm{degrees}$.

- **Inverse trigonometric [radians]**: `asin`, `acos`, `atan`, `asec`, `acsc`, `acot` represent $\arcsin(x)$, $\arccos(x)$, $\arctan(x)$, $\mathrm{arcsec}(x)$, $\mathrm{arccsc}(x)$, $\mathrm{arccot}(x)$, respectively, where the result is given in $\mathrm{radians}$.

- **Inverse trigonometric [degrees]**: `asind`, `acosd`, `atand`, `asecd`, `acscd`, `acotd` represent $\arcsin(x)$, $\arccos(x)$, $\arctan(x)$, $\mathrm{arcsec}(x)$, $\mathrm{arccsc}(x)$, $\mathrm{arccot}(x)$, respectively, where the result is given in $\mathrm{degrees}$.

- **Special trigonometric [radians]**: `sincos`, `sinpi`, `cospi`, `sincospi`, `sinc`, `cosc` represent tuple $(\sin(x), \cos(x))$, $\sin(\pi x)$, $\cos(\pi x)$, tuple $(\sin(\pi x), \cos(\pi x))$, $\mathrm{sinc}(x) = \frac{\sin(\pi x)}{\pi x}$, $\mathrm{cosc}(x) = \frac{\mathrm{d}}{\mathrm{d}x}\mathrm{sinc}(x)$, respectively.

- **Hyperbolic trigonometric [radians]**: `sinh`, `cosh`, `tanh`, `sech`, `csch`, `coth` represent $\sinh(x)$, $\cosh(x)$, $\tanh(x)$, $\mathrm{sech}(x) = \frac{1}{\cosh(x)}$, $\mathrm{csch}(x) = \frac{1}{\sinh(x)}$, $\coth(x) = \frac{1}{\tanh(x)}$, respectively, where x is given in $\mathrm{radians}$.

- **Inverse hyperbolic [radians]**: `asinh`, `acosh`, `atanh`, `asech`, `acsch`, `acoth` represent $\mathrm{arcsinh}(x)$, $\mathrm{arccosh}(x)$, $\mathrm{arctanh}(x)$, $\mathrm{arcsech}(x)$, $\mathrm{arccsch}(x)$, $\mathrm{arccoth}(x)$, respectively, where x is given in $\mathrm{radians}$.

- **Angle conversion**: `deg2rad` , `rad2deg` represent $x \to \frac{2\pi \cdot x}{360}$ and $x \to \frac{360 \cdot x}{2\pi}$, respectively.
- **Logarithmic/exponential**: `log` , `log2` , `log10` , `log1p` , `exp` , `exp2` , `exp10` , `expm1` represent $\ln(x) = \log_e(x)$, $\log_2(x)$, $\lg(x) = \log_{10}(x)$, $\ln(1+x)$, $\exp(x)$, 2^x, 10^x, and $\exp(x) - 1$, respectively. `log1p` is more accurate than `log` when the argument is close to 1 in value.
- **Roots**: `sqrt` , `cbrt` represent $\sqrt{x}$ (with an error if x is real and negative) and $\sqrt[3]{x}$, respectively.
- **Absolute values**: `abs` , `abs2` represent $|x|$ and $|x|^2$, respectively. `sign` returns zero if x==0 and x/|x| otherwise (i.e., ±1 for real x).
- **Complex numbers**: with $j = \sqrt{-1}$ and $z = x + y \cdot j$, functions `real` , `imag` , `reim` , `conj` , `angle` , `cis` , `cispi` return x, y, tuple (x, y), $\bar{z} = x - y \cdot j$, $\arctan\left(\frac{y}{x}\right)$, $\exp(z \cdot j)$, $\exp(2\pi z \cdot j)$, respectively.
- **Combinatorics**: `factorial(n)` returns $n!$
- **Rational numbers**: `numerator(x)` returns the numerator of rational number `x` , `denominator(x)` returns the denominator of rational number `x`

In [655…

```
sin(pi/2)
```

```
1.0
```

In [656…

```
sind(90)
```

```
1.0
```

In [657…

```
sincos(pi/3)
```

```
(0.8660254037844386, 0.5000000000000001)
```

In [658…

```
rad2deg(pi/2)
```

```
90.0
```

In [659…

```
log10(10)
```

```
1.0
```

In [660…

```
cbrt(8)
```

```
2.0
```

In [661…

```
abs2(-2)
```

```
4
```

In [662…
```
sign(-2), sign(0)
```
(-1, 0)

In [663…
```
real(2+3im)
```
2

In [664…
```
reim(2+3im)
```
(2, 3)

In [665…
```
conj(2+3im)
```
2 - 3im

In [666…
```
angle(2+3im)
```
0.982793723247329

In [667…
```
rad2deg(angle(2+3im))
```
56.309932474020215

In [668…
```
cis(2+3im)
```
-0.02071873100224288 + 0.045271253156092976im

In [669…
```
exp(im*(2+3im))
```
-0.02071873100224288 + 0.045271253156092976im

In [670…
```
cispi(2+3im)
```
8.069951757030463e-5 + 0.0im

In [671…
```
exp(pi*im*(2+3im))
```
8.069951757030463e-5 - 1.976568117704183e-20im

In [672…
```
factorial(1),factorial(2),factorial(3),factorial(4)
```
(1, 2, 6, 24)

In [673…
```
denominator(22//7)
```
7

Basic mathematical functions: two or more arguments

The following *multiple argument* mathematical functions are given in the on-line Julia documentation at https://julialang.org/:

- **Geometry**: `hypot` [hypotenuse] represents $(x_1, \ldots, x_n) \to \sqrt{\sum_1^n |x_i|^2}$, base `b` logarithm of `x`. For a complex number `z`, `hypot(z)` is `hypot(re(z),im(z))`.
- **Logarithm**: `log(b,x)` represents the logarithm of `x` at base `b`, i.e., $\log_b(x) = \frac{\log_c(x)}{\log_c(b)}$
- **Modulo, etc.**: `div(x,y)` (or: `÷(x,y)`, `x÷y`) represents `x/y` truncated to an integer; `mod(x,y)` represents `x` modulo `y`; `rem(x,y)` (or: `%(x,y)`, `x%y`) returns the remainder from Euclidian division; `rem2pi(x)` returns `rem(x,pi)`; `mod2pi(x)` returns `mod(x,pi)`.
- **Integer**: `gcd(x1,...,xn)` returns greatest common divisor of arguments, `lcm(x1,...,xn)` returns least common multiple.
- **Combinatorics**: `binomial(n,k)` returns $\binom{n}{k}$ where n and k are integers.
- **Min-max**: `min(x, y, ...)` returns the minimum of the arguments; `max(x, y, ...)` returns the maximum of the arguments; `minmax(x, y)` returns the tuple of min and max of the arguments; `clamp(x, lo, hi)` returns the saturation of `x` in the interval `[lo,hi]`.

In [674…
```
hypot(2,3), hypot(2+3im)
```
```
(3.605551275463989, 3.605551275463989)
```

In [675…
```
log(3,8), log2(8)/log2(3)
```
```
(1.892789260714372, 1.8927892607143724)
```

In [676…
```
div(14,3), div(14.,3)
```
```
(4, 4.0)
```

In [677…
```
mod(14,3)
```
```
2
```

In [678…
```
rem(14,3)
```
```
2
```

In [679…
```
div(14,3)*3 + mod(14,3)
```
```
14
```

In [680…
```
gcd(12,36,45)
```
```
3
```

In [681…

```
lcm(12,36,45)
```

```
180
```

In [682…

```
binomial(12,12)
```

```
1
```

In [683…

```
binomial(12,9)
```

```
220
```

In [684…

```
min(-1,0,-2,3)
```

```
-2
```

In [685…

```
max(-1,0,-2,3)
```

```
3
```

In [686…

```
clamp(1,-1,3)
```

```
1
```

In [687…

```
clamp(-2,-1,3)
```

```
-1
```

In [688…

```
clamp(4,-1,3)
```

```
3
```

Operation on numbers

Floating point deconstruction

- **Mantissa**: `significand(x)` extracts the mantissa of `x` in binary representation, with value of the same type as `x` in the interval $[1, 2)$
- **Exponent**: `exponent(x)` returns the exponent of a normalized floating point number (i.e., with mantissa in $[1, 2)$). The result is the largest integer `y` such that `2ʸ ≤ abs(x)` .

Rounding

- **Rounding**: `round(x)` rounds `x` to an integer of the same type as `x` ; `round(x;digits=val)` where `val` is an integer, returns `x` rounded to `val` digits; `round(x;sigdigits=val)` where `val` is an integer returns `x` rounded to `val` significant digits.

- **Ceiling**: `ceil(x)` returns the nearest integral value of the same type as `x` that is greater than or equal to `x` ; `ceil(x;digits=val)` where `val` is an integer, returns the nearest integral value of the same type as `x` that is greater than or equal to `x` to `val` digits; `ceil(x;sigdigits=val)` where `val` is an integer, returns the nearest integral value of the same type as `x` that is greater than or equal to `x` to `val` significant digits.

- **Floor**: `floor(x)` returns the nearest integral value of the same type as `x` that is less than or equal to `x`; `floor(x;digits=val)` where `val` is an integer, returns the nearest integral value of the same type as `x` that is less than or equal to `x` to `val` digits; `floor(x;sigdigits=val)` where `val` is an integer, returns the nearest integral value of the same type as `x` that is less than or equal to `x` to `val` significant digits.

- **Trunc**: `trunc(x)` returns the nearest integral value of the same type as `x` whose *absolute value* is less than or equal to `x`; `trunc(x;digits=val)` where `val` is an integer, returns the nearest integral value of the same type as `x` whose absolute value is less than or equal to `x` to `val` digits; `trunc(x;sigdigits=val)` where `val` is an integer, returns the nearest integral value of the same type as `x` whose absolute value is less than or equal to `x` to `val` significant digits.

Rational approximation

- **Rational numbers**: `rationalize(x)` with `x` a floating point number returns a rational number approximately equal to `x`, `rationalize(x;tol=val)` returns a rational approximation of `x` to tolerance given by `val`, `rationalize(T,x)` returns a rational number with numerator/denominator given by type `T`.

BigFloat precision

- **BigFloat**: `BigFloat(x;precision=val)` produces a `BigFloat` type of `x`, with `val` (integer) number of digits in the mantissa (significand). Function `precision(x)` checks the precision of `x`.

In [689…
```
x=100pi
```
```
314.1592653589793
```

In [690…
```
s_x = significand(x)
```
```
1.227184630308513
```

In [691…
```
e_x = exponent(x)
```
```
8
```

In [692…
```
s_x*2^e_x
```
```
314.1592653589793
```

In [693…
```
round(x)
```

314.0

In [694…

```
round(x,digits=3)
```

314.159

In [695…

```
round(x,sigdigits=4)
```

314.2

In [696…

```
ceil(x),ceil(-x)
```

(315.0, -314.0)

In [697…

```
floor(x),floor(-x)
```

(314.0, -315.0)

In [698…

```
trunc(x),trunc(-x)
```

(314.0, -314.0)

In [699…

```
ceil(-x)
```

-314.0

In [700…

```
floor(-x)
```

-315.0

In [701…

```
trunc(-x)
```

-314.0

In [702…

```
rationalize(Float64(pi)), rationalize(Float64(pi);tol=1e-3), rationalize(Int16,Float64(
```

(165707065//52746197, 201//64, 355//113, 22//7)

In [703…

```
x = BigFloat(pi;precision=12)
```

3.1416

In [704…

```
significand(x)
```

1.57080078125