Functions

August 12, 2021

1 Functions

1.1 Arguments

Julia functions support both positional arguments and default values:

[1]:
```
function draw_face(x, y, width=3, height=4)
    println("x=$x, y=$y, width=$width, height=$height")
end

draw_face(10, 20, 30)
```

```
x=10, y=20, width=30, height=4
```

However, unlike in Python, positional arguments SHOULD NOT be named when the function is called:

[2]:
```
try
    draw_face(10, 20, width=30)
catch ex
    ex
end
```

[2]:
```
MethodError(var"#draw_face##kw"(), ((width = 30,), draw_face, 10, 20),
0x00000000000074de)
```

Julia also supports a variable number of arguments (called "varargs") using operator slurping and splatting (e.g., `arg...`).

This is the equivalent of Python's `*arg`:

[3]:
```
function copy_files(target_dir, paths...)
    println("target_dir=$target_dir, paths=$paths")
end

copy_files("/tmp", "a.txt", "b.txt")
```

```
target_dir=/tmp, paths=("a.txt", "b.txt")
```

Keyword arguments are supported, after a semicolon `;` (`;` is not required when the function is called; however, a good practice is to be included):

[4]:
```
function copy_files2(paths...; confirm=false, target_dir)
    println("paths=$paths, confirm=$confirm, $target_dir")
end

copy_files2("a.txt", "b.txt"; target_dir="/tmp")
```

```
paths=("a.txt", "b.txt"), confirm=false, /tmp
```

Notes: * `target_dir` has no default value, so it is a required argument. * `confimr` has a default value; it is an optiona argument. * The order of the keyword arguments does not matter. * The arguments cannot be repeated in the function calls (an error messagge is produced)

You can have another operator slurping and splatting in the keyword section.

It corresponds to Python's `**kwargs`:

[5]:
```
function copy_files3(paths...; confirm=false, target_dir, options...)
    println("paths=$paths, confirm=$confirm, $target_dir")
    verbose = options[:verbose]
    println("verbose=$verbose")
    println("provided options are $(options...))")
end

copy_files3("a.txt", "b.txt"; target_dir="/tmp", verbose=true, timeout=60)
```

```
paths=("a.txt", "b.txt"), confirm=false, /tmp
verbose=true
provided options are Pair{Symbol, Integer}(:verbose, true)Pair{Symbol,
Integer}(:timeout, 60))
```

The `options` is a dictionary.

The dictionary keys are **symbols**, e.g., `:verbose`.

Symbols are like strings, less flexible but faster. Symbols are typically used as keys or identifiers. cc

Julia	Python (3.8+ if `/` is used)
`function foo(a, b=2, c=3)`	`...endfoo(1, 2) # positional only`
`function foo(;a=1, b, c=3)`	`...endfoo(c=30, b=2) # keyword only`
`function foo(a, b=2; c=3, d)`	`...endfoo(1; d=4) # pos only; then keyword only`
`function foo(a, b=2, c...)`	`...endfoo(1, 2, 3, 4) # positional only`
`function foo(a, b=1, c...; d=1, e, f...)`	`...endfoo(1, 2, 3, 4, e=5, x=10, y=20)`

1.2 Concise Functions

In Julia, the following definition:

[6]:
```
square(x) = x^2
```

[6]:
```
square (generic function with 1 method)
```

is equivalent to:

[7]:
```
function square(x)
    x^2
end
```

[7]:
```
square (generic function with 1 method)
```

For example, here's a shorter way to define the `estimate_pi()` function in Julia:

[8]:
```
estimate_pi3(n) = 4 * sum((isodd(i) ? -1 : 1)/(2i+1) for i in 0:n)
```

[8]:
```
estimate_pi3 (generic function with 1 method)
```

To define a function on one line in Python, you need to use a `lambda` (but this is generally frowned upon, since the resulting function's name is `"<lambda>"`; very original name!):

```
# PYTHON
square = lambda x: x**2
assert square.__name__ == "<lambda>"
```

1.3 Anonymous Functions

Just like in Python, you can define anonymous functions:

[9]:
```
map(x -> x^2, 1:4)
```

[9]:
```
4-element Vector{Int64}:
  1
  4
  9
 16
```

Here is the equivalent Python code:

```
list(map(lambda x: x**2, range(1, 5)))
```

Notes: * `map()` returns an array in Julia, instead of an iterator like in Python. * You could use a comprehension Julia as well: `[x^2 for x in 1:4]`.

Julia	Python
`x -> x^2`	`lambda x: x**2`
`(x,y) -> x + y`	`lambda x,y: x + y`
`() -> println("yes")`	`lambda: print("yes")`

In Python, lambda functions must be simple expressions. They cannot contain multiple statements.

In Julia, they can be as long as you want. Indeed, you can create a multi-statement block using the syntax `(stmt_1; stmt_2; ...; stmt_n)`. The return value is the output of the last statement. For example:

[10]:
```
map(x -> (println("Number $x"); x^2), 1:4)
```

```
Number 1
Number 2
Number 3
Number 4
```

[10]:
```
4-element Vector{Int64}:
  1
  4
  9
 16
```

This syntax can span multiple lines:

[11]:
```
map(x -> (
  println("Number $x");
  x^2), 1:4)
```

```
Number 1
Number 2
Number 3
Number 4
```

[11]:
```
4-element Vector{Int64}:
  1
  4
  9
 16
```

But in this case, it's probably clearer to use the `begin ... end` syntax instead:

[12]:
```
map(x -> begin
        println("Number $x")
        x^2
    end, 1:4)
```

```
Number 1
Number 2
Number 3
Number 4
```

[12]:
```
4-element Vector{Int64}:
  1
  4
  9
 16
```

Notice that this syntax allows you to drop the semicolons `;` at the end of each line in the block.

Yet another way to define an anonymous function is using the `function (args) ... end` syntax:

[13]:
```
map(function (x)
        println("Number $x")
        x^2
    end, 1:4)
```

```
Number 1
Number 2
Number 3
Number 4
```

[13]:
```
4-element Vector{Int64}:
  1
  4
  9
 16
```

Lastly, if you're passing the anonymous function as the first argument to a function (as is the case in this example), it's usually much preferable to define the anonymous function immediately after the function call, using the `do` syntax, like this:

[14]:
```
map(1:4) do x
  println("Number $x")
  x^2
end
```

```
Number 1
Number 2
Number 3
Number 4
```

[14]:
```
4-element Vector{Int64}:
  1
  4
  9
 16
```

This syntax lets you easily define constructs that feel like language extensions:

[15]:
```
function my_for(func, collection)
    for i in collection
        func(i)
    end
end

my_for(1:4) do i
    println("The square of $i is $(i^2)")
end
```

```
The square of 1 is 1
The square of 2 is 4
```

```
The square of 3 is 9
The square of 4 is 16
```

In fact, Julia has a similar `foreach()` function.

The `do` syntax could be used to write a Domain Specific Language (DSL), for example an infrastructure automation DSL:

[16]:
```
function spawn_server(startup_func, server_type)
    println("Starting $server_type server")
    server_id = 1234
    println("Configuring server $server_id...")
    startup_func(server_id)
end

# This is the DSL part
spawn_server("web") do server_id
    println("Creating HTML pages on server $server_id...")
end
```

```
Starting web server
Configuring server 1234…
Creating HTML pages on server 1234…
```

It's also quite nice for event-driven code:

[17]:
```
handlers = []

on_click(handler) = push!(handlers, handler)

click(event) = foreach(handler->handler(event), handlers)

on_click() do event
    println("Mouse clicked at $event")
end

on_click() do event
    println("Beep.")
end

click((x=50, y=20))
click((x=120, y=10))
```

```
Mouse clicked at (x = 50, y = 20)
Beep.
Mouse clicked at (x = 120, y = 10)
Beep.
```

It can also be used to create context managers, for example to automatically close an object after it has been used, even if an exception is raised:

[18]:
```
function with_database(func, name)
    println("Opening connection to database $name")
    db = "a db object for database $name"
    try
        func(db)
    finally
        println("Closing connection to database $name")
    end
end

with_database("jobs") do db
    println("I'm working with $db")
    #error("Oops") # try uncommenting this line
end
```

```
Opening connection to database jobs
I'm working with a db object for database jobs
Closing connection to database jobs
```

The equivalent code in Python would look like this:

```
# PYTHON
class Database:
    def __init__(self, name):
        self.name = name
    def __enter__(self):
        print(f"Opening connection to database {self.name}")
        return f"a db object for database {self.name}"
    def __exit__(self, type, value, traceback):
        print(f"Closing connection to database {self.name}")

with Database("jobs") as db:
    print(f"I'm working with {db}")
    #raise Exception("Oops") # try uncommenting this line
```

Or you could use `contextlib`:

```
from contextlib import contextmanager

@contextmanager
def database(name):
    print(f"Opening connection to database {name}")
    db = f"a db object for database {name}"
    try:
        yield db
    finally:
        print(f"Closing connection to database {name}")

with database("jobs") as db:
    print(f"I'm working with {db}")
```

```
#raise Exception("Oops") # try uncommenting this line
```

1.4 Piping

If you are used to the Object Oriented syntax `"a b c".upper().split()`, you may feel that writing `split(uppercase("a b c"))` is a bit backwards. If so, the piping operation `|>` is for you:

[19]:
```
"a b c" |> uppercase |> split
```

[19]:
```
3-element Vector{SubString{String}}:
 "A"
 "B"
 "C"
```

If you want to pass more than one argument to some of the functions, you can use anonymous functions:

[20]:
```
"a b c" |> uppercase |> split |> tokens->join(tokens, ", ")
```

[20]:
```
"A, B, C"
```

The dotted version of the pipe operator works as you might expect, applying the ith function of the right array to the ith value in the left array:

[21]:
```
[/2, "hello", 4] .|> [sin, length, x->x^2]
```

[21]:
```
3-element Vector{Real}:
  1.0
  5
 16
```

1.5 Composition

Julia also lets you compose functions like mathematicians do, using the composition operator (`\circ<tab>` in the REPL or Jupyter):

[22]:
```
f = exp ∘ sin ∘ sqrt
f(2.0) == exp(sin(sqrt(2.0)))
```

[22]:
```
true
```

2 Methods

Earlier, we discussed structs, which look a lot like Python classes, with instance variables and constructors, but they did not contain any methods (just the inner constructors). In Julia, methods are defined separately, like regular functions:

[23]:
```
struct Person
    name
    age
```

```
end

function greetings(greeter)
    println("Hi, my name is $(greeter.name), I am $(greeter.age) years old.")
end

p = Person("Alice", 70)
greetings(p)
```

```
Hi, my name is Alice, I am 70 years old.
```

Since the `greetings()` method in Julia is not bound to any particular type, we can use it with any other type we want, as long as that type has a `name` and an `age` (i.e., if it quacks like a duck):

[24]:
```
struct City
    name
    country
    age
end

using Dates
c = City("Auckland", "New Zealand", year(now()) - 1840)

greetings(c)
```

```
Hi, my name is Auckland, I am 181 years old.
```

You could code this the same way in Python if you wanted to:

```
# PYTHON
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

class City:
    def __init__(self, name, country, age):
        self.name = name
        self.country = country
        self.age = age

def greetings(greeter):
    print(f"Hi there, my name is {greeter.name}, I am {greeter.age} years old.")

p = Person("Lucy", 70)
greetings(p)

from datetime import date
c = City("Auckland", "New Zealand", date.today().year - 1840)
greetings(c)
```

However, many Python programmers would use inheritance in this case:

```
class Greeter:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def greetings(self):
        print(f"Hi there, my name is {self.name}, I am {self.age} years old.")

class Person(Greeter):
    def __init__(self, name, age):
        super().__init__(name, age)

class City(Greeter):
    def __init__(self, name, country, age):
        super().__init__(name, age)
        self.country = country

p = Person("Lucy", 70)
p.greetings()

from datetime import date
c = City("Auckland", "New Zealand", date.today().year - 1840)
c.greetings()
```

2.1 Extending a Function

One nice thing about having a class hierarchy is that you can override methods in subclasses to get specialized behavior for each class. For example, in Python you could override the `greetings()` method like this:

```
# PYTHON
class Developer(Person):
    def __init__(self, name, age, language):
        super().__init__(name, age)
        self.language = language
    def greetings(self):
        print(f"Hi there, my name is {self.name}, I am {self.age} years old.")
        print(f"My favorite language is {self.language}.")

d = Developer("Amy", 40, "Julia")
d.greetings()
```

Notice that the expression `d.greetings()` will call a different method if `d` is a `Person` or a `Developer`. This is called "polymorphism": the same method call behaves differently depending on the type of the object. The language chooses which actual method implementation to call, based on the type of `d`: this is called method "dispatch". More specifically, since it only depends on a single variable, it is called "single dispatch".

The good news is that Julia can do single dispatch as well:

[25]:
```
struct Developer
    name
    age
    language
end

function greetings(dev::Developer)
    println("Hi, my name is $(dev.name), I am $(dev.age) years old.")
    println("My favorite language is $(dev.language).")
end

d = Developer("Amy", 40, "Julia")
greetings(d)
```

```
Hi, my name is Amy, I am 40 years old.
My favorite language is Julia.
```

Notice that the `dev` argument is followed by `::Developer`, which means that this method will only be called if the argument has that type.

We have **extended** the `greetings` **function**, so that it now has two different implementations, called **methods**, each for different argument types: namely, `greetings(dev::Developer)` for arguments of type `Developer`, and `greetings(greeter)` for values of any other type.

You can easily get the list of all the methods of a given function:

[26]:
```
methods(greetings)
```

[26]:
```
# 2 methods for generic function "greetings":
[1] greetings(dev::Developer) in Main at In[25]:7
[2] greetings(greeter) in Main at In[23]:6
```

You can also get the list of all the methods which take a particular type as argument:

[27]:
```
methodswith(Developer)
```

[27]:
```
[1] greetings(dev::Developer) in Main at In[25]:7
```

When you call the `greetings()` function, Julia automatically dispatches the call to the appropriate method, depending on the type of the argument. If Julia can determine at compile time what the type of the argument will be, then it optimizes the compiled code so that there's no choice to be made at runtime. This is called **static dispatch**, and it can significantly speed up the program. If the argument's type can't be determined at compile time, then Julia makes the choice at runtime, just like in Python: this is called **dynamic dispatch**.

2.2 Multiple Dispatch

Julia actually looks at the types of *all* the positional arguments, not just the first one. This is called **multiple dispatch**. For example:

[28]:
```
multdisp(a::Int64, b::Int64) = 1
multdisp(a::Int64, b::Float64) = 2
multdisp(a::Float64, b::Int64) = 3
multdisp(a::Float64, b::Float64) = 4

multdisp(10, 20) # try changing the arguments to get each possible output
```

[28]:
```
1
```

Julia always chooses the most specific method it can, so the following method will only be called if the first argument is neither an `Int64` nor a `Float64`:

[29]:
```
multdisp(a::Any, b::Int64) = 5

multdisp("10", 20)
```

[29]:
```
5
```

Julia will raise an exception if there is some ambiguity as to which method is the most specific:

[30]:
```
ambiguity(a::Int64, b) = 1
ambiguity(a, b::Int64) = 2

try
    ambiguity(10, 20)
catch ex
    ex
end
```

[30]:
```
MethodError(ambiguity, (10, 20), 0x0000000000007508)
```

To solve this problem, you can explicitely define a method for the ambiguous case:

[31]:
```
ambiguity(a::Int64, b::Int64) = 3
ambiguity(10, 20)
```

[31]:
```
3
```

So you can have polymorphism in Julia, just like in Python. This means that you can write your algorithms in a generic way, without having to know the exact types of the values you are manipulating, and it will work fine, as long as these types act in the general way you expect (i.e., if they "quack like ducks"). For example:

[32]:
```
function how_can_i_help(greeter)
    greetings(greeter)
    println("How can I help?")
end

how_can_i_help(p) # called on a Person
how_can_i_help(d) # called on a Developer
```

```
Hi, my name is Alice, I am 70 years old.
How can I help?
Hi, my name is Amy, I am 40 years old.
My favorite language is Julia.
How can I help?
```

2.3 Calling `super()`?

You may have noticed that the `greetings(dev::Developer)` method could be improved, since it currently duplicates the implementation of the base method `greetings(greeter)`. In Python, you would get rid of this duplication by calling the base class's `greetings()` method, using `super()`:

```python
# PYTHON
class Developer(Person):
    def __init__(self, name, age, language):
        super().__init__(name, age)
        self.language = language
    def greetings(self):
        super().greetings() # <== THIS!
        print(f"My favorite language is {self.language}.")

d = Developer("Amy", 40, "Julia")
d.greetings()
```

In Julia, you can do something pretty similar, although you have to implement your own `super()` function, as it is not part of the language:

[33]:
```julia
super(dev::Developer) = Person(dev.name, dev.age)

function greetings(dev::Developer)
    greetings(super(dev))
    println("My favorite language is $(dev.language).")
end

greetings(d)
```

```
Hi, my name is Amy, I am 40 years old.
My favorite language is Julia.
```

However, this implementation creates a new `Person` instance when calling `super(dev)`, copying the `name` and `age` fields. That's okay for small objects, but it's not ideal for larger ones. Instead, you can explicitely call the specific method you want by using the `invoke()` function:

[34]:
```julia
function greetings(dev::Developer)
    invoke(greetings, Tuple{Any}, dev)
    println("My favorite language is $(dev.language).")
end

greetings(d)
```

```
Hi, my name is Amy, I am 40 years old.
My favorite language is Julia.
```

The `invoke()` function expects the following arguments: * The first argument is the function to call. * The second argument is the type of the desired method's arguments tuple: `Tuple{TypeArg1, TypeArg2, etc.}`. In this case we want to call the base function, which takes a single `Any` argument (the `Any` type is implicit when no type is specified). * Lastly, it takes all the arguments to be passed to the method. In this case, there's just one: `dev`.

As you can see, we managed to get the same advantages Object-Oriented programming offers, without defining classes or using inheritance. This takes a bit of getting used to, but you might come to prefer this style of generic programming. Indeed, OO programming encourage you to bundle data and behavior together, but this is not always a good idea. Let's look at one example:

```python
# PYTHON
class Rectangle:
    def __init__(self, height, width):
        self.height = height
        self.width = width
    def area(self):
        return self.height * self.width

class Square(Rectangle):
    def __init__(self, length):
        super().__init__(length, length)
```

It makes sense for the `Square` class to be a subclass of the `Rectangle` class, since a square **is a** special type of rectangle. It also makes sense for the `Square` class to inherit from all of the `Rectangle` class's behavior, such as the `area()` method. However, it does not really make sense for rectangles and squares to have the same memory representation: a `Rectangle` needs two numbers (`height` and `width`), while a `Square` only needs one (`length`).

It's possible to work around this issue like this:

```python
# PYTHON
class Rectangle:
    def __init__(self, height, width):
        self.height = height
        self.width = width
    def area(self):
        return self.height * self.width

class Square(Rectangle):
    def __init__(self, length):
        self.length = length
    @property
    def width(self):
        return self.length
    @property
    def height(self):
        return self.length
```

That's better: now, each square is only represented using a single number. We've inherited the behavior, but not the data.

In Julia, you could code this like so:

[35]:
```
struct Rectangle
    width
    height
end

width(rect::Rectangle) = rect.width
height(rect::Rectangle) = rect.height

area(rect) = width(rect) * height(rect)

struct Square
    length
end

width(sq::Square) = sq.length
height(sq::Square) = sq.length
```

[35]:
```
height (generic function with 2 methods)
```

[36]:
```
area(Square(5))
```

[36]:
```
25
```

Notice that the `area()` function relies on the getters `width()` and `height()`, rather than directly on the fields `width` and `height`. In this way, the argument can be of any type at all, as long as it has these getters.

2.4 Abstract Types

One nice thing about the class hierarchy we defined in Python is that it makes it clear that a square **is a** kind of rectangle. Any new function you define that takes a `Rectangle` as an argument will automatically accept a `Square` as well, but no other non-rectangle type. In contrast, our `area()` function currently accepts anything at all.

In Julia, a concrete type like `Square` cannot extend another concrete type like `Rectangle`. However, any type can extend from an abstract type. Let's define some abstract types to create a type hierarchy for our `Square` and `Rectangle` types.

[37]:
```
abstract type AbstractShape end
abstract type AbstractRectangle <: AbstractShape end  # <: means "subtype of"
abstract type AbstractSquare <: AbstractRectangle end
```

The `<:` operator means "subtype of".

Now we can attach the `area()` function to the `AbstractRectangle` type, instead of any type at all:

[38]:
```
area(rect::AbstractRectangle) = width(rect) * height(rect)
```

[38]:
```
area (generic function with 2 methods)
```

Now we can define the concrete types, as subtypes of `AbstractRectangle` and `AbstractSquare`:

[39]:
```
struct Rectangle_v2 <: AbstractRectangle
  width
  height
end

width(rect::Rectangle_v2) = rect.width
height(rect::Rectangle_v2) = rect.height

struct Square_v2 <: AbstractSquare
  length
end

width(sq::Square_v2) = sq.length
height(sq::Square_v2) = sq.length
```

[39]:
```
height (generic function with 4 methods)
```

In short, the Julian approach to type hierarchies looks like this:

- Create a hierarchy of abstract types to represent the concepts you want to implement.
- Write functions for these abstract types. Much of your implementation can be coded at that level, manipulating abstract concepts.
- Lastly, create concrete types, and write the methods needed to give them the behavior that is expected by the generic algorithms you wrote.

This pattern is used everywhere in Julia's standard libraries. For example, here are the supertypes of `Float64` and `Int64`:

[40]:
```
Base.show_supertypes(Float64)
```

```
Float64 <: AbstractFloat <: Real <: Number <: Any
```

[41]:
```
Base.show_supertypes(Int64)
```

```
Int64 <: Signed <: Integer <: Real <: Number <: Any
```

Note: Julia implicitly runs `using Core` and `using Base` when starting the REPL. However, the `show_supertypes()` function is not exported by the `Base` module, thus you cannot access it by just typing `show_supertypes(Float64)`. Instead, you have to specify the module name: `Base.show_supertypes(Float64)`.

And here is the whole hierarchy of `Number` types:

[42]:
```
function show_hierarchy(root, indent=0)
    println(repeat(" ", indent * 4), root)
```

```
    for subtype in subtypes(root)
        show_hierarchy(subtype, indent + 1)
    end
end

show_hierarchy(Number)
```

```
Number
    Complex
    Real
        AbstractFloat
            BigFloat
            Float16
            Float32
            Float64
        AbstractIrrational
            Irrational
        FixedPointNumbers.FixedPoint
            FixedPointNumbers.Fixed
            FixedPointNumbers.Normed
        Integer
            Bool
            Signed
                BigInt
                Int128
                Int16
                Int32
                Int64
                Int8
            Unsigned
                UInt128
                UInt16
                UInt32
                UInt64
                UInt8
        Rational
        Ratios.SimpleRatio
        StatsBase.PValue
        StatsBase.TestStat
```

2.5 Iterator Interface

You will sometimes want to provide a way to iterate over your custom types. In Python, this requires defining the `__iter__()` method which should return an object which implements the `__next__()` method. In Julia, you must define at least two functions: * `iterate(::YourIteratorType)`, which must return either `nothing` if there are no values in the sequence, or `(first_value, iterator_state)`. * `iterate(::YourIteratorType, state)`, which must return either `nothing` if there are no more values, or `(next_value, new_iterator_state)`.

For example, let's create a simple iterator for the Fibonacci sequence:

[43]:
```
struct FibonacciIterator end
```

[44]:
```
import Base.iterate

iterate(f::FibonacciIterator) = (1, (1, 1))

function iterate(f::FibonacciIterator, state)
    new_state = (state[2], state[1] + state[2])
    (new_state[1], new_state)
end
```

[44]:
```
iterate (generic function with 375 methods)
```

Now we can iterate over a `FibonacciIterator` instance:

[45]:
```
for f in FibonacciIterator()
    println(f)
    f > 10 && break
end
```

```
1
1
2
3
5
8
13
```

2.6 Indexing Interface

You can also create a type that will be indexable like an array (allowing syntax like `a[5] = 3`). In Python, this requires implementing the `__getitem__()` and `__setitem__()` methods. In Julia, you must implement the `getindex(A::YourType, i)`, `setindex!(A::YourType, v, i)`, `firstindex(A::YourType)` and `lastindex(A::YourType)` methods.

[46]:
```
struct MySquares end

import Base.getindex, Base.firstindex

getindex(::MySquares, i) = i^2
firstindex(::MySquares) = 0

S = MySquares()
S[10]
```

[46]:
```
100
```

[47]:
```
S[begin]
```

[47]:
```
0
```

[48]:
```
getindex(S::MySquares, r::UnitRange) = [S[i] for i in r]
```

[48]:
```
getindex (generic function with 342 methods)
```

[49]:
```
S[1:4]
```

[49]:
```
4-element Vector{Int64}:
  1
  4
  9
 16
```

For more details on these interfaces, and to learn how to build full-blown array types with broadcasting and more, check out this page.

2.7 Creating a Number Type

Let's create a `MyRational` struct and try to make it mimic the built-in `Rational` type:

[50]:
```
struct MyRational <: Real
    num # numerator
    den # denominator
end
```

[51]:
```
MyRational(2, 3)
```

[51]:
```
MyRational(2, 3)
```

It would be more convenient and readable if we could type `2  3` to create a `MyRational`:

[52]:
```
function  (num, den)
    MyRational(num, den)
end
```

[52]:
```
  (generic function with 1 method)
```

[53]:
```
2  3
```

[53]:
```
MyRational(2, 3)
```

I chose because it's a symbol that Julia's parser treats as a binary operator, but which is otherwise not used by Julia (see the full list of parsed symbols and their priorities). This particular symbol will have the same priority as multiplication and division.

If you want to know how to type it and check that it is unused, type in Juia REPL `?` (copy/paste the symbol).

The question mark `?` switches the REPL into a help mode.

Now let's make it possible to add two `MyRational` values. We want it to be possible for our `MyRational` type to be used in existing algorithms which rely on `+`, so we must create a new method for the `Base.+` function:

[54]:
```
import Base.+

function +(r1::MyRational, r2::MyRational)
    (r1.num * r2.den + r1.den * r2.num)   (r1.den * r2.den)
end
```

[54]:
```
+ (generic function with 295 methods)
```

[55]:
```
2   3 + 3   5
```

[55]:
```
MyRational(19, 15)
```

It's important to import `Base.+` first, or else you would just be defining a new `+` function in the current module (`Main`), which would not be called by existing algorithms.

You can easily implement `*`, `^` and so on, in much the same way.

Let's change the way `MyRational` values are printed, to make them look a bit nicer. For this, we must create a new method for the `Base.show(io::IO, x)` function:

[56]:
```
import Base.show

function show(io::IO, r::MyRational)
    print(io, "$(r.num)   $(r.den)")
end

2   3 + 3   5
```

[56]:
```
19   15
```

We can expand the `show()` function so it can provide an HTML representation for `MyRational` values. This will be called by the `display()` function in Jupyter or Colab:

[57]:
```
function show(io::IO, ::MIME"text/html", r::MyRational)
    print(io, "<sup><b>$(r.num)</b></sup>&frasl;<sub><b>$(r.den)</b></sub>")
end

2   3 + 3   5
```

[57]:
```
19   15
```

Next, we want to be able to perform any operation involving `MyRational` values and values of other `Number` types. For example, we may want to multiply integers and `MyRational` values. One option is to define a new method like this:

[58]:
```
import Base.*

function *(r::MyRational, i::Integer)
    (r.num * i)  r.den
end

2  3 * 5
```

[58]: 10 3

Since multiplication is commutative, we need the reverse method as well:

[59]:
```
function *(i::Integer, r::MyRational)
    r * i # this will call the previous method
end

5 * (2  3) # we need the parentheses since * and  have the same priority
```

[59]: 10 3

It's cumbersome to have to define these methods for every operation. There's a better way, which we will explore in the next two sections.

2.8 Conversion

It is possible to provide a way for integers to be automatically converted to `MyRational` values:

[60]:
```
import Base.convert

MyRational(x::Integer) = MyRational(x, 1)

convert(::Type{MyRational}, x::Integer) = MyRational(x)

convert(MyRational, 42)
```

[60]: 42 1

The `Type{MyRational}` type is a special type which has a single instance: the `MyRational` type itself. So this `convert()` method only accepts `MyRational` itself as its first argument (and we don't actually use the first argument, so we don't even need to give it a name in the function declaration).

Now integers will be automatically converted to `MyRational` values when you assign them to an array whose element type if `MyRational`:

[61]:
```
a = [2  3] # the element type is MyRational
a[1] = 5    # convert(MyRational, 5) is called automatically
push!(a, 6) # convert(MyRational, 6) is called automatically
println(a)
```

```
MyRational[5  1, 6  1]
```

Conversion will also occur automatically in these cases: * `r::MyRational = 42`: assigning an integer to `r` where `r` is a local variable with a declared type of `MyRational`. * `s.b = 42` if `s` is a struct and `b` is a field of type `MyRational` (also when calling `new(42)` on that struct, assuming `b` is the first field). * `return 42` if the return type is declared as `MyRational` (e.g., `function f(x)::MyRational ... end`).

However, there is no automatic conversion when calling functions:

[62]:
```
function for_my_rationals_only(x::MyRational)
    println("It works:", x)
end

try
    for_my_rationals_only(42)
catch ex
    ex
end
```

[62]:
```
MethodError(for_my_rationals_only, (42,), 0x000000000000752f)
```

2.9 Promotion

The `Base` functions `+`, `-`, `*`, `/`, `^`, etc. all use a "promotion" algorithm to convert the arguments to the appropriate type. For example, adding an integer and a float promotes the integer to a float before the addition takes place. These functions use the `promote()` function for this. For example, given several integers and a float, all integers get promoted to floats:

[63]:
```
promote(1, 2, 3, 4.0)
```

[63]:
```
(1.0, 2.0, 3.0, 4.0)
```

This is why a sum of integers and floats results in a float:

[64]:
```
1 + 2 + 3 + 4.0
```

[64]:
```
10.0
```

The `promote()` function is also called when creating an array. For example, the following array is a `Float64` array:

[65]:
```
a = [1, 2, 3, 4.0]
```

[65]:
```
4-element Vector{Float64}:
 1.0
 2.0
 3.0
 4.0
```

What about the `MyRational` type? Rather than create new methods for the `promote()` function, the recommended approach is to create a new method for the `promote_rule()` function. It takes two types and returns the type to convert to:

[66]:
```
promote_rule(Float64, Int64)
```

[66]:
```
Float64
```

Let's implement a new method for this function, to make sure that any subtype of the `Integer` type will be promoted to `MyRational`:

[67]:
```
import Base.promote_rule

promote_rule(::Type{MyRational}, ::Type{T}) where {T <: Integer} = MyRational
```

[67]:
```
promote_rule (generic function with 162 methods)
```

This method definition uses **parametric types**: the type `T` can be any type at all, as long as it is a subtype of the `Integer` abstract type. If you tried to define the method `promote_rule(::Type{MyRational}, ::Type{Integer})`, it would expect the type `Integer` itself as the second argument, which would not work, since the `promote_rule()` function will usually be called with concrete types like `Int64` as its arguments.

Let's check that it works:

[68]:
```
promote(5, 2  3)
```

[68]:
```
(5  1, 2  3)
```

Yep! Now whenever we call `+`, `-`, etc., with an integer and a `MyRational` value, the integer will get automatically promoted to a `MyRational` value:

[69]:
```
5 + 2  3
```

[69]:
```
17  3
```

Under the hood: * this called `+(5, 2  3)`, * which called the `+(::Number, ::Number)` method (thanks to multiple dispatch), * which called `promote(5, 2  3)`, * which called `promote_rule(Int64, MyRational)`, * which called `promote_rule(::MyRational, ::T) where {T <: Integer}`, * which returned `MyRational`, * then the `+(::Number, ::Number)` method called `convert(MyRational, 5)`, * which called `MyRational(5)`, * which returned `MyRational(5, 1)`, * and finally `+(::Number, ::Number)` called `+(MyRational(5, 1), MyRational(2, 3))`, * which returned `MyRational(17, 3)`.

The benefit of this approach is that we only need to implement the `+`, `-`, etc. functions for pairs of `MyRational` values, not with all combinations of `MyRational` values and integers.

If your head hurts, it's perfectly normal. ;-) Writing a new type that is easy to use, flexible and plays nicely with existing types takes a bit of planning and work, but the point is that you will not write these every day, and once you have, they will make your life much easier.

Now let's handle the case where we want to execute operations with `MyRational` values and floats. In this case, we naturally want to promote the `MyRational` value to a float. We first need to define how to convert a `MyRational` value to any subtype of `AbstractFloat`:

[70]:
```
convert(::Type{T}, x::MyRational) where {T <: AbstractFloat} = T(x.num / x.den)
```

[70]:
```
convert (generic function with 496 methods)
```

This `convert()` works with any type `T` which is a subtype of `AbstractFloat`. It just computes `x.num / x.den` and converts the result to type `T`. Let's try it:

[71]:
```
convert(Float64, 3//2)
```

[71]:
```
1.5
```

Now let's define a `promote_rule()` method which will work for any type `T` which is a subtype of `AbstractFloat`, and which will give priority to `T` over `MyRational`:

[72]:
```
promote_rule(::Type{MyRational}, ::Type{T}) where {T <: AbstractFloat} = T
```

[72]:
```
promote_rule (generic function with 163 methods)
```

[73]:
```
promote(1//2, 4.0)
```

[73]:
```
(0.5, 4.0)
```

Now we can combine floats and `MyRational` values easily:

[74]:
```
2.25 ^ (1//2)
```

[74]:
```
1.5
```

2.10 Parametric Types and Functions

Julia's `Rational` type is actually a **parametric type** which ensures that the numerator and denominator have the same type `T`, subtype of `Integer`. Here's a new version of our rational struct which enforces the same constraint:

[75]:
```
struct MyRational2{T <: Integer}
    num::T
    den::T
end
```

To instantiate this type, we can specify the type `T`:

[76]:
```
MyRational2{BigInt}(2, 3)
```

[76]:
```
MyRational2{BigInt}(2, 3)
```

Alternatively, we can use the `MyRational2` type's default constructor, with two integers of the same type:

[77]:
```
MyRational2(2, 3)
```

[77]:
```
MyRational2{Int64}(2, 3)
```

If we want to be able to construct a `MyRational2` with integers of different types, we must write an appropriate constructor which handles the promotion rule:

[78]:
```
function MyRational2(num::Integer, den::Integer)
    MyRational2(promote(num, den)...)
end
```

[78]:
```
MyRational2
```

This constructor accepts two integers of potentially different types, and promotes them to the same type. Then it calls the default `MyRational2` constructor which expects two arguments of the same type. The syntax `f(args...)` is analog to Python's `f(*args)`.

Let's see if this works:

[79]:
```
MyRational2(2, BigInt(3))
```

[79]:
```
MyRational2{BigInt}(2, 3)
```

Note that all parametrized types such as `MyRational2{Int64}` or `MyRational2{BigInt}` are subtypes of `MyRational2`. So if a function accepts a `MyRational2` argument, you can pass it an instance of any specific, parametrized type:

[80]:
```
function for_any_my_rational2(x::MyRational2)
    println(x)
end

for_any_my_rational2(MyRational2{BigInt}(1, 2))
for_any_my_rational2(MyRational2{Int64}(1, 2))
```

```
MyRational2{BigInt}(1, 2)
MyRational2{Int64}(1, 2)
```

A more explicit (but verbose) syntax for this function is:

[81]:
```
function for_any_my_rational2(x::MyRational2{T}) where {T <: Integer}
    println(x)
end
```

[81]:
```
for_any_my_rational2 (generic function with 1 method)
```

It's useful to think of types as sets. For example, the `Int64` type represents the set of all 64-bit integer values, so `42 isa Int64`: * When `x` is an instance of some type `T`, it is an element of the set `T` represents, and `x isa T`. * When `U` is a subtype of `V`, `U` is a subset of `V`, and `U <: V`.

The `MyRational2` type itself (without any parameter) represents the set of all values of `MyRational2{T}` for all subtypes `T` of `Integer`. In other words, it is the union of all the `MyRational2{T}` types. This is called a `UnionAll` type, and indeed the type `MyRational2` itself is an instance of the `UnionAll` type:

[82]:
```
@assert MyRational2{BigInt}(2, 3) isa MyRational2{BigInt}
@assert MyRational2{BigInt}(2, 3) isa MyRational2
@assert MyRational2 === (MyRational2{T} where {T <: Integer})
@assert MyRational2{BigInt} <: MyRational2
```

```
@assert MyRational2 isa UnionAll
```

If we dump the `MyRational2` type, we can see that it is a `UnionAll` instance, with a parameter type `T`, constrained to a subtype of the `Integer` type (since the upper bound `ub` is `Integer`):

[83]:
```
dump(MyRational2)
```

```
UnionAll
  var: TypeVar
    name: Symbol T
    lb: Union{}
    ub: Integer <: Real
  body: MyRational2{T<:Integer} <: Any
    num::T
    den::T
```

3 Macros

All macros start with an `@` sign: `@which`, `@assert`, `@time`, `@benchmark`, `@btime` and `@doc`.

Macro is a function which can fully inspect the expression that follows it, and apply any transformation to that code at parse time, before compilation.

This makes it possible for anyone to effectively extend the language in any way they please. Whereas C/C++ macros just do simple text replacement, **Julia macros are powerful metaprogramming tools**.

On the flip side, this also means that **each macro has its own syntax and behavior**. And they can be dangerous and difficult to comprehand.

Here's a simple macro that replaces `a + b` expressions with `a - b`, and leaves other expressions alone.

[84]:
```
macro addtosub(x)
  if x.head == :call && x.args[1] == :+ && length(x.args) == 3
    Expr(:call, :-, x.args[2], x.args[3])
  else
    x
  end
end

@addtosub 10 + 2
```

[84]:
```
8
```

In this macro definition, `:call`, `:+` and `:-` are **symbols**. These are similar to strings, only more efficient and less flexible. They are typically used as identifiers, such as keys in dictionaries.

If you're curious, the macro works because the parser converts `10 + 2` to `Expr(:call, :+, 10, 2)` and passes this expression to the macro (before compilation). The `if` statement checks that the expression is a function call, where the called function is the `+` function, with two arguments. If so,

then the macro returns a new expression, corresponding to a call to the `-` function, with the same arguments. So `a + b` becomes `a - b`.