Notes 10: Other topics

Chris Paciorek

2025-02-25

Table of contents

Interacting with other languages

Student presentations cover interaction with Python and Julia.

Calling out to C/C++

We can call C (and presumably C++) functions.

Here's an example in which we'll call the `dnorm4` function in the R math library, which is the C function used by R's `dnorm` function, which returns the log density.

```
x = 0
mu = 10
sd = 1
dev = @ccall "/usr/lib/libRmath.so".dnorm4(x::Cdouble,mu::Cdouble,sd::Cdouble)::Cdouble;
print(dev)
```

```
-50.918938533204674
```

Perhaps the use of C code related to R is a bit confusing, but the key things here are to point `@ccall` to a shared object/dynamically linked library file (.so on Linux/MacOS, presumably .dll on Windows) that contains the compiled code for the function you want to use and to provide input and output types as indicated above.

To investigate further, we'd want to look into passing pointers to allow for handling arrays.

Meta programming

Julia has powerful functionality for using code to actually manipulate and run Julia code. This is called meta programming (sometimes also called *computing on the language*) and is something one can also do in R. I'm not sure how much of this one can easily do in Python.

We'll just crack the surface here.

Manipulating variables

```
num = 3
varname = "x" * string(num)
var = Meta.parse(varname)  # One way to create the variable as a `Symbol` object.
var
```

```
:x3
```

```
var2 = Symbol(varname)  #  Another way.
typeof(var)
```

```
Symbol
```

```
value = 7

mycode = quote
  $var = $value
end

x3
```

```
LoadError: UndefVarError: `x3` not defined
UndefVarError: `x3` not defined
```

```
mycode
```

```
quote
    #= In[5]:4 =#
    x3 = 7
end
```

```
value = 23;
eval(mycode);
x3
```

```
7
```

```
value = 9;
mycode = quote
  $var = value
end
```

```
mycode
```

```
quote
    #= In[8]:3 =#
    x3 = value
end
```

```
value = 23;
eval(mycode);
x3
```

```
23
```

Manipulating code expressions / ASTs

We can create and manipulate code expressions.

```
textcode = "a+b"
expr1 = Meta.parse(textcode)

expr2 = :(a+b)
expr3 = Expr(:call, :+, :a, :b)

print(expr1 == expr2)
print(expr1 == expr3)
typeof(expr1)
```

```
truetrue
```

```
Expr
```

```
expr1.args
```

```
3-element Vector{Any}:
 :+
 :a
 :b
```

```
a=7
b=3
eval(expr1)
```

```
10
```

Let's modify the expression!

```
expr1.args[1] = :*
print(expr1)
```

```
a * b
```

```
eval(expr1)
```

```
21
```

Remember ASTs (abstract syntax trees?

```
expr = :(tan(x) + 3 * sin(x))
expr.args
```

```
3-element Vector{Any}:
 :+
 :(tan(x))
 :(3 * sin(x))
```

```
expr.args[3]
```

```
:(3 * sin(x))
```

We have access to the full AST!

```
expr.args[3].args[2]
```

```
3
```

And we could modify the AST as shown earlier.

Here's how assignment is captured:

```
mycode = :(c = a + b)

mycode.head
```

```
:(=)
```

```
mycode.args[1]
```

```
:c
```

```
mycode.args[2]
```

```
:(a + b)
```

More on interpolation/substitution

We can control when variables are treated as symbols versus evaluated (as already seen).

Use of `$` allows us to "interpolate" values into the expression (as we've seen with printing in Notes 1).

```
a = 7;
expr2 = :($a+b)                 # `$a` interpolates. `b` is code.
expr3 = Expr(:call, :+, a, :b)  # `a` is evaluated. `:b` is code.
expr2
```

```
:(7 + b)
```

```
expr3
```

```
:(7 + b)
```

```
a = 55
b = 999
eval(expr2)
```

```
1006
```

```
eval(expr3)
```

```
1006
```

Being able to control what is treated as code and what is treated as values is quite powerful when manipulating code expressions.

Use of `quote`

We can use `quote` to create an object containing multiple expressions:

```
mycode = quote
  a + b
  tan(x) + 3 * sin(x)
end

mycode.args[2] == :(a+b)
```

```
true
```

```
mycode.args[4] == :(tan(x) + 3 * sin(x))
```

```
true
```

Use cases

It might not be obvious where this would be useful. One situation is if you were writing some sort of code translator, such as a compiler of some sort. Another (relatedly) is in writing macros.

For example here's a macro that creates a version of a function that reports how many times it has been called.

```
const CALL_COUNTS = Dict{Any, Int}()

import ExprTools

# Macro to create a counted version of a function
macro counted(func)
    def = ExprTools.splitdef(func)
    name = def[:name]
    body = def[:body]
```

```
    # Create new function definition that increments counter.
    def[:body] =  quote
        CALL_COUNTS[$name] = get(CALL_COUNTS, $name, 0) + 1
        $body
    end

    return esc(ExprTools.combinedef(def))
end

@counted function my_add(x, y)
    return x + y
end

# Use the function
my_add(2, 3)
my_add(4, 5)

CALL_COUNTS[my_add]
```