15.9 Automatically Reindent Code

15.9.1 Problem

You want to reformat your code so that it lines up nicely and is indented consistently.

15.9.2 Solution

One of the many features of the RStudio IDE is it helps with routine code manipulation, such as reformatting. To get consistent indentation, highlight the text you'd like reindented, press `Ctrl+i` on Win or Linux or press `Cmd+i` on Mac.

15.9.3 Discussion

When editing code it's easy to end up with indentation that is inconsistent and a little confusing. Like the following code, for example:

```
for (i in 1:5) {
if(i>=3) {
print(i**2)
       }
else print(i*3)
           }
```

While that's valid code, it can be a little tricky to read because of the odd indentation. If we highlight the text in RStudio IDE and press `Ctrl+i` (or `Cmd+i` on Mac) then our code gets consistent indentation:

```
for (i in 1:5) {
  if(i>=3) {
    print(i**2)
  }
  else print(i*3)
}
```

15.9.4 See Also

Links to the RStudio cheatsheets are availiable inside the RStudio IDE by clicking help -> Cheatsheets or by going directly to https://www.rstudio.com/resources/cheatsheets/

15.10 Executing R in Parallel Locally

15.10.1 Problem

You have code that takes a while to run and you would like to speed it up by using more of the cores on your local computer.

15.10.2 Solution

The easiest solution to get up and running with is to use the `furrr` package which, in turn uses the `future` package to provide parallel processing via functions that feel like the functions from `purrr` except that they operate in parallel.

To use `furrr` we downloaded the latest deveopment version from Github because the package is still under active development:

```
devtools::install_github("DavisVaughan/furrr")
```

To use `furrr` to parallize our code we call the `furrr::future_map` in place of the `purrr::map` function we discussed in the recipe Applying a Function to Each List Element {#recipe-id149} from chapter 6. But first we have to tell `furrr` how we want to parallelize. In this case we want a `multiprocess` parallel process which uses all our local processors. So we set that up by calling `plan(multiprocess)`. Then we can apply a function to every element in our list using `future_map`:

```
library(furrr)

plan(multiprocess)

future_map(my_list, some_function)
```

15.10.3 Discussion

Let's do an example simulation to illustrate parallelization. A classic stochastic simulation is to draw random points inside of a 2 x 2 box and see how many points fall within 1 unit from the center of the box. The ratio of points inside the box / total points then multiplied by 4 is a good estimate of *pi*. The following function takes one input, `n_iterations` which is the number of random points to simulate. Then it returns the resulting average estimate of *pi*:

```
simulate_pi <- function(n_iterations) {
  rand_draws <- matrix(runif(2 * n_iterations, -1, 1),  ncol = 2)
  num_in <- sum(sqrt(rand_draws[,1]**2 + rand_draws[,2]**2) <= 1)
  pi_hat <- (num_in / n_iterations) * 4
  return(pi_hat)
}
simulate_pi(1000000)
#> [1] 3.1409
```

As you can see above, even with 1,000,000 simulations the result is only accurate out to a couple of decimal points. This is not a very efficient way to estimate *pi* but it works for our illustration.

For the purpose of comparision later, let's run 200 runs of this *pi* simulator where each run has 2,500,000 simulated points. We'll do this by creating a list with 200 elements, each of which is the value 2,500,000 which we will pass to `simulate_pi`. We'll time the code with the `tictoc` package:

```
library(purrr)     # for `map`
library(tictoc)    # for timing our code

draw_list <- as.list(rep(5000000, 200))

tic('simulate pi - single process')
sims_list <- map(draw_list, simulate_pi)
toc()
#> simulate pi - single process: 96.702 sec elapsed
```

```
mean(unlist(sims_list))
#> [1] 3.14163
```

That runs in less than two minutes and gives an estimate of *pi* based on a billion simulation (5m * 200).

Now let's take the exact same process but run it through `future_map` to run it in parallel.

```
library(furrr)
#> Loading required package: future
plan(multiprocess)

tic('simulate pi - parallel')
sims_list <- future_map(draw_list, simulate_pi)
toc()
#> simulate pi - parallel: 33.306 sec elapsed
mean(unlist(sims_list))
#> [1] 3.14154
```

The example above was run on a Macbook Pro with 4 physical cores and 2 virtual cores per physical core. When running code in parallel the best case scenario is that the run time is reduced by 1/(number of physical cores). With 4 physical cores you can see the parallel run time is much faster than the single threaded version but not quite $1/4^{th}$ the runtime of the single threaded version. There is always some overhead from moving the data around so you will never experience the best case scenario. And the more data each iteration produces, the less speed improvement you will experience from parallelization.

15.10.4 See Also

Executing R in Parallel Remotely {#recipe-parallel_remote}

15.11 Executing R in Parallel Remotely

15.11.1 Problem

You have access to a number of remote machines and you would like to run your code in parallel across them all.

15.11.2 Solution

Running code in parallel across multiple machines can be tricky to get set up initially. However if we start with a few key prerequisites in place, the process has a much higher probability of success.

Starting prerequisites: 1. You can ssh from your main machine to each remote node without a password using previously generated ssh keys 2. The remote nodes all have R installed (ideally the same version of R) 3. Paths are set such that you can run `Rscript` from SSH: ##JDL give example 4. The remote nodes have the package `furrr` installed (which in turn installs `future`) 5. The remote nodes already have all the packages installed on which your distributed code depends.

Once you have worker nodes that are set up and ready to go, you can create a cluster by calling `makeClusterPSOCK` from the `future` package. Then use the resulting cluster with the `furrr` function `future_map`:

```
library(furrr) # loads future as a dependency

workers <- c('node_1.domain.com', 'node_2.domain.com')

cl <- makeClusterPSOCK(
  worker = workers)

plan(cluster, workers = cl)

future_map(my_list, some_function)
```

15.11.3 Discussion

We have two big Linux machines named `von-neumann12` and `von-neumann15` which we can use to run numerical models. These machines meet the criteria listed above so they are good candidates to be our backend for a furrr/future cluster. Let's do the same pi simulation we did in the prior recipe using the `simulate_pi` function:

```
library(tidyverse)
library(furrr)
library(tictoc)

my_workers <- c('von-neumann12','von-neumann15')

cl <- makeClusterPSOCK(
  workers = my_workers,
  rscript = '/home/jal/anaconda2/bin/Rscript', ##JDL fix path for ssh on server
  verbose=TRUE
)

draw_list <- as.list(rep(5000000, 200))

plan(cluster, workers = cl)

tic('simulate pi - parallel map')
sims_list_parallel <- draw_list %>%
  future_map(simulate_pi)
toc()
#> simulate pi - parallel map: 116.986 sec elapsed

mean(unlist(sims_list_parallel))
#> [1] 3.14167
```

The two nodes in our ad hoc cluster each have 32 processors and 128GB of RAM. But if you compare the run time of the above code with the run time of the prior recipe run on a humble Macbook Pro, you'll notice that the Macbook executed the code in about the same time as the multi-CPU linux cluster with 64 total processors! This unintuitive surprise happens because the code above only runs on one CPU per each cluster node. So, as a result, it only uses two CPUs while the Macbook example uses all 4 of its CPUs.

So how do we run parallel code on a cluster and have each node also run in parallel across multiple CPU cores? To do that we need to make three changes to our code:

1. Create a nested parallel plan that uses **both** `cluster` and `multiprocess`

2. Create an input list which is a nested list… each cluster machine will get an item from the main list which contains sublist items that it can process in parallel across all its CPUs.
3. Call `future_map` twice using a nested call. The outer `future_map` will parallel items across the cluster nodes, then the inner call will parallelize across the CPUs.

To created the nested parallel plan, we will create a multi-part plan by passing a list of two plans to the `plan` function like this: `plan(list(tweak(cluster, workers = cl), multiprocess))`

The second change is to create the nested list to iterate on. We can do that by using the `split` command and passing it our prior list followed by a vector of `1:4` like so: `split(draw_list, 1:4)` This will break the initial list into 4 sublists. So our resulting list will have 4 elements. Each sublist will have 50 inputs for our final `simulate_pi` function.

The third change to our code is to create a nested `future_map` call that will pass each of our 4 list elements to the worker nodes and then the worker nodes will iterate over the elements of each sublist. We create that nested function like this: `future_map(draw_list, ~future_map(.,simulate_pi))` The `~` sets up R to expect an anonymous function inside the first `future_map` call and the `.` tells R where the put the list element. The ananomous function in this example is a seperate call to `future_map` which gets executed on each node.

Here's all three changes integrated into code:

```
# nested parallel plan. The first part of the plan is the cluster call
# followed by the multiprocess
plan(list(tweak(cluster, workers = cl), multiprocess))

## test this to make sure the named list thing works
# break the draw_list into a nested list with fewer elements
draw_list_nested <- split(draw_list, 1:4)

tic('simulate pi - parallel nested map')
sims_list_nested_parallel <- future_map(draw_list_nested, ~future_map(.,simulate_pi))
toc()
#> simulate pi - parallel nested map: 15.964 sec elapsed
mean(unlist(sims_list_nested_parallel))
#> [1] 3.14158
```

You can see the run time decreased substantially from the prior example. Although with 32 processors on each node, we're not seeing a 32x improvement in run time. This is because we're only passing 50 sets of simulations to each node. Each node runs 32 sets of simulations in the first pass then in the second pass runs only 18, leaving half the CPUs idle.

Let's keep the CPUs a little more busy by increasing our total simulations from 1 billion to 25 billion. Then we'll break them into 500 work blocks to be spread to the two worker nodes:

```
draw_list <- as.list(rep(5000000, 5000))
draw_list_nested <- split(draw_list, 1:50)

plan(list(tweak(cluster, workers = cl), multiprocess))

tic('simulate pi - parallel nested map')
sims_list_nested_parallel <- future_map(draw_list_nested, ~future_map(.,simulate_pi))
toc()
#> simulate pi - parallel nested map: 260.532 sec elapsed
mean(unlist(sims_list_nested_parallel))
#> [1] 3.14157
```

15.11.4 See Also

The `future` package has multiple excellent vignettes. For understanding the nested `plan` call, start with `vignette('future-3-topologies',package='future')`

Further info about `furrr` can be found at its Github page: https://github.com/DavisVaughan/furrr