

Starting out with R and RStudio

ModuleI: 2025_IntroR_and_RStudio

2025-09-01

Contents

Welcome to Omics Science!	29
What we will cover:	29
How is this course different?	31
1 Getting Started with R and RStudio	32
1.1 A Beginner’s Guide for First-Year Graduate Students	32
1.2 What You’re Installing and Why	32
1.3 Step 1: Install R	32
1.4 Step 2: Install Additional Tools (macOS Only)	33
1.5 Step 3: Install Git	33
1.6 Step 4: Install RStudio	34
1.7 Step 5: Test Your Installation	34
1.8 Step 6: Download Course Materials	35
1.9 Step 7: Install Required R Packages	37
1.10 Step 8: Verify Everything Works	37
1.11 What Success Looks Like	38
1.12 Common Problems and Solutions	39
1.13 Getting Help	39
1.14 Next Steps	39
2 Introduction to R and RStudio	41
2.1 What is R?	41
2.2 What is RStudio?	41

CONTENTS	3
2.3 Open a project directory in RStudio	42
2.4 RStudio Interface	42
2.5 Interacting with R	43
2.6 The R syntax	44
2.7 Assignment operator	44
2.8 Variables	45
2.9 Interacting with data in R	46
3 R syntax and data structures	48
3.1 Data Types	48
3.2 Data Structures	48
4 Functions in R	54
4.1 Functions and their arguments	54
5 Packages and libraries	57
5.1 Packages and Libraries	57
6 Subsetting: vectors and factors	61
6.1 Selecting data using indices	61
7 Reading and data inspection	65
7.1 Reading data into R	65
7.2 Inspecting data structures	68
8 Dataframes and matrices	70
8.1 Dataframes	70
9 Logical operators for matching	76
9.1 Logical operators to match elements	76
9.2 The %in% operator	76
10 Reordering to match datasets	81
10.1 Reordering data to match	81

10.2 The <code>match</code> function	82
11 Plotting and data visualization	86
11.1 Dataframe setup for visualization	86
12 Plotting with ggplot2	89
12.1 Data Visualization with <code>ggplot2</code>	89
13 Boxplot with ggplot2: exercise	97
13.1 Generating a Boxplot with <code>ggplot2</code>	97
14 Saving data and plots to file	100
14.1 Writing data to file	100
14.2 Exporting figures to file	100
15 Finding help	102
15.1 Asking for help	102
16 Tidyverse data wrangling	105
16.1 Tidyverse basics	106
16.2 [1] 9.110434	107
16.3 [1] 9.11	107
16.4 [1] 9.11	107
16.5 Experimental data	108
16.6 Analysis goal and workflow	110
16.7 # A tibble: 3,644 x 14	113
16.8 query.number significant p.value term.size query.size overlap.size recall	113
16.9	113
16.101 1 TRUE 0.00434 111 5850 52 0.009	113
16.112 1 TRUE 0.0033 110 5850 52 0.009	113
16.123 1 TRUE 0.0297 39 5850 21 0.004	113
16.134 1 TRUE 0.0193 70 5850 34 0.006	113
16.145 1 TRUE 0.0148 26 5850 16 0.003	113

16.156 1 TRUE 0.0187 22 5850 14 0.002	113
16.167 1 TRUE 0.0226 48 5850 25 0.004	113
16.178 1 TRUE 0.0491 17 5850 11 0.002	113
16.189 1 TRUE 0.00798 164 5850 71 0.012	113
16.1910 1 TRUE 0.0439 19 5850 12 0.002	113
16.20# i 3,634 more rows	113
16.21# i 7 more variables: precision , term.id , domain ,	113
16.22# subgraph.number , term.name , relative.depth ,	113
16.23# intersection	113
16.24# A tibble: 6 x 7	115
16.25term.id term.name p.value query.size term.size overlap.size intersection	115
16.26	115
16.271 GO:0032606 type I inte~ 0.00434 5850 111 52 rnf216,polr~	115
16.282 GO:0032479 regulation ~ 0.0033 5850 110 52 rnf216,polr~	115
16.293 GO:0032480 negative re~ 0.0297 5850 39 21 rnf216,otud~	115
16.304 GO:0032481 positive re~ 0.0193 5850 70 34 polr3b,polr~	115
16.315 GO:0010644 cell commun~ 0.0148 5850 26 16 pkp2,prkaca~	115
16.326 GO:0086064 cell commun~ 0.0187 5850 22 14 pkp2,prkaca~	115
16.33Next steps	118
17 Codebook for IntroR and RStudio	120
17.1 01 Setting up R and RStudio	120
17.2 02 Introduction to R and RStudio	120
17.3 [1] 8	120
17.4 [1] 8	121
17.5 [1] 8	121
17.6 03 R syntax and data structures	121
17.7 [1] 4.6 3000.0 50000.0	122
17.8 [1] “ecoli” “human” “corn”	122
17.9 [1] “low” “high” “medium” “high” “low” “medium” “high”	122

17.10[1] low high medium high low medium high	122
17.11Levels: high low medium	122
17.12[1] 7	122
17.13[1] “high” “low” “medium”	123
17.14species glengths	123
17.151 ecoli 4.6	123
17.162 human 3000.0	123
17.173 corn 50000.0	123
17.1804 Functions in R	124
17.19[1] 9	124
17.20[1] 2.144761 54.772256 223.606798	124
17.21[1] 3	124
17.22[1] 3.14	124
17.23[1] 400	124
17.24function (x, digits = 0, ...)	125
17.25NULL	125
17.26	127
17.27round> round(.5 + -2:4) # IEEE / IEC rounding: -2 0 0 2 2 4 4	127
17.28[1] -2 0 0 2 2 4 4	127
17.29	127
17.30round> ## (this is <i>good</i> behaviour – do <i>NOT</i> report it as bug !)	127
17.31round>	127
17.32round> (x1 <- seq(-2, 4, by = .5))	127
17.33[1] -2.0 -1.5 -1.0 -0.5 0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0	127
17.34	127
17.35round> round(x1) #- IEEE / IEC rounding !	127
17.36[1] -2 -2 -1 0 0 0 1 2 2 2 3 4 4	127
17.37	127
17.38round> x1[trunc(x1) != floor(x1)]	127
17.39[1] -1.5 -0.5	127

17.40	127
17.41 round> x1[round(x1) != floor(x1 + .5)]	127
17.42[1] -1.5 0.5 2.5	127
17.43	127
17.44 round> (non.int <- ceiling(x1) != floor(x1))	127
17.45[1] FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE	127
17.46[13] FALSE	127
17.47	127
17.48 round> x2 <- pi * 100^(-1:3)	127
17.49	127
17.50 round> round(x2, 3)	127
17.51[1] 0.031 3.142 314.159 31415.927 3141592.654	127
17.52	127
17.53 round> signif(x2, 3)	127
17.54[1] 3.14e-02 3.14e+00 3.14e+02 3.14e+04 3.14e+06	127
17.55[1] 25	128
17.56 05 Packages and libraries	129
17.57 R version 4.5.1 (2025-06-13)	131
17.58 Platform: x86_64-apple-darwin20	131
17.59 Running under: macOS Sequoia 15.5	131
17.60	131
17.61 Matrix products: default	131
17.62 BLAS: /Library/Frameworks/R.framework/Versions/4.5-x86_64/Resources/lib/libRblas.0.dylib	131
17.63 LAPACK: /Library/Frameworks/R.framework/Versions/4.5-x86_64/Resources/lib/libRlapack.dylib; LAPACK version 3.12.1	131
17.64	131
17.65 locale:	131
17.66[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8	131
17.67	131
17.68 time zone: America/Los_Angeles	131

17.69tzcode source: internal	131
17.70	131
17.71attached base packages:	131
17.72[1] stats graphics grDevices utils datasets methods base	131
17.73	131
17.74other attached packages:	131
17.75[1] lubridate_1.9.4 forcats_1.0.0 stringr_1.5.1 dplyr_1.1.4	131
17.76[5] purrr_1.1.0 readr_2.1.5 tidyverse_1.3.1 tibble_3.3.0	131
17.77[9] ggplot2_3.5.2 tidyverse_2.0.0	131
17.78	131
17.79loaded via a namespace (and not attached):	131
17.80[1] bit_4.6.0 gtable_0.3.6 crayon_1.5.3 compiler_4.5.1	131
17.81[5] tidyselect_1.2.1 parallel_4.5.1 scales_1.4.0 yaml_2.3.10	131
17.82[9] fastmap_1.2.0 R6_2.6.1 generics_0.1.4 knitr_1.50	131
17.83[13] bookdown_0.44 pillar_1.11.0 RColorBrewer_1.1-3 tzdb_0.5.0	131
17.84[17] rlang_1.1.6 utf8_1.2.6 stringi_1.8.7 xfun_0.53	131
17.85[21] bit64_4.6.0-1 timechange_0.3.0 cli_3.6.5 withr_3.0.2	131
17.86[25] magrittr_2.0.3 formatR_1.14 digest_0.6.37 grid_4.5.1	131
17.87[29] vroom_1.6.5 rstudioapi_0.17.1 hms_1.1.3 lifecycle_1.0.4	131
17.88[33] vctrs_0.6.5 evaluate_1.0.4 glue_1.8.0 farver_2.1.2	131
17.89[37] fansi_1.0.6 rmarkdown_2.29 tools_4.5.1 pkgconfig_2.0.3	131
17.90[41] htmltools_0.5.8.1	131
17.91[1] “GlobalEnv” “package:lubridate” “package:forcats”	132
17.92[4] “package:stringr” “package:dplyr” “package:purrr”	132
17.93[7] “package:readr” “package:tidyverse” “package:tibble”	132
17.94[10] “package:ggplot2” “package:tidyverse” “package:stats”	132
17.95[13] “package:graphics” “package:grDevices” “package:utils”	132
17.96[16] “package:datasets” “package:methods” “Autoloads”	132
17.97[19] “package:base”	132
17.98[1] “%+%” “%+replace%”	135

17.99[3] “aes” “aes_”	135
17.10[5] “aes_all” “aes_auto”	135
17.10[7] “aes_q” “aes_string”	135
17.10[9] “after_scale” “after_stat”	135
17.10[11] “alpha” “annotate”	135
17.10[13] “annotation_custom” “annotation_logticks”	135
17.10[15] “annotation_map” “annotation_raster”	135
17.10[17] “arrow” “as_label”	135
17.10[19] “as_labeller” “autolayer”	135
17.10[21] “autoplot” “AxisSecondary”	135
17.10[23] “benchplot” “binned_scale”	135
17.11[25] “borders” “calc_element”	135
17.11[27] “check_device” “combine_vars”	135
17.11[29] “continuous_scale” “Coord”	135
17.11[31] “coord_cartesian” “coord_equal”	135
17.11[33] “coord_fixed” “coord_flip”	135
17.11[35] “coord_map” “coord_munch”	135
17.11[37] “coord_polar” “coord_quickmap”	135
17.11[39] “coord_radial” “coord_sf”	135
17.11[41] “coord_trans” “CoordCartesian”	135
17.11[43] “CoordFixed” “CoordFlip”	135
17.12[45] “CoordMap” “CoordPolar”	135
17.12[47] “CoordQuickmap” “CoordRadial”	135
17.12[49] “CoordSf” “CoordTrans”	135
17.12[51] “cut_interval” “cut_number”	135
17.12[53] “cut_width” “datetime_scale”	135
17.12[55] “derive” “diamonds”	135
17.12[57] “discrete_scale” “draw_key_abline”	135
17.12[59] “draw_key_blank” “draw_key_boxplot”	135
17.12[61] “draw_key_crossbar” “draw_key_dotplot”	135

17.12[63] “draw_key_label” “draw_key_linerange”	135
17.13[65] “draw_key_path” “draw_key_point”	135
17.13[67] “draw_key_pointrange” “draw_key_polygon”	135
17.13[69] “draw_key_rect” “draw_key_smooth”	135
17.13[71] “draw_key_text” “draw_key_timeseries”	135
17.13[73] “draw_key_vline” “draw_key_vpath”	135
17.13[75] “dup_axis” “economics”	135
17.13[77] “economics_long” “el_def”	135
17.13[79] “element_blank” “element_grob”	135
17.13[81] “element_line” “element_rect”	135
17.13[83] “element_render” “element_text”	135
17.14[85] “enexpr” “enexprs”	135
17.14[87] “enquo” “enquos”	135
17.14[89] “ensym” “ensyms”	135
17.14[91] “expand_limits” “expand_scale”	135
17.14[93] “expansion” “expr”	135
17.14[95] “Facet” “facet_grid”	135
17.14[97] “facet_null” “facet_wrap”	135
17.14[99] “FacetGrid” “FacetNull”	135
17.14[101] “FacetWrap” “faithful”	135
17.14[103] “fill_alpha” “find_panel”	135
17.15[105] “flip_data” “flipped_names”	135
17.15[107] “fortify” “Geom”	135
17.15[109] “geom_abline” “geom_area”	135
17.15[111] “geom_bar” “geom_bin_2d”	135
17.15[113] “geom_bin2d” “geom_blank”	135
17.15[115] “geom_boxplot” “geom_col”	135
17.15[117] “geom_contour” “geom_contour_filled”	135
17.15[119] “geom_count” “geom_crossbar”	135
17.15[121] “geom_curve” “geom_density”	135

17.15[123] “geom_density_2d” “geom_density_2d_filled”	135
17.16[125] “geom_density2d” “geom_density2d_filled”	135
17.16[127] “geom_dotplot” “geom_errorbar”	135
17.16[129] “geom_errorbarh” “geom_freqpoly”	135
17.16[131] “geom_function” “geom_hex”	135
17.16[133] “geom_histogram” “geom_hline”	135
17.16[135] “geom_jitter” “geom_label”	135
17.16[137] “geom_line” “geom_linerange”	135
17.16[139] “geom_map” “geom_path”	135
17.16[141] “geom_point” “geom_pointrange”	135
17.16[143] “geom_polygon” “geom_qq”	135
17.17[145] “geom_qq_line” “geom_quantile”	135
17.17[147] “geom_raster” “geom_rect”	135
17.17[149] “geom_ribbon” “geom_rug”	135
17.17[151] “geom_segment” “geom_sf”	135
17.17[153] “geom_sf_label” “geom_sf_text”	135
17.17[155] “geom_smooth” “geom_spoke”	135
17.17[157] “geom_step” “geom_text”	135
17.17[159] “geom_tile” “geom_violin”	135
17.17[161] “geom_vline” “GeomAbline”	135
17.17[163] “GeomAnnotationMap” “GeomArea”	135
17.18[165] “GeomBar” “GeomBlank”	135
17.18[167] “GeomBoxplot” “GeomCol”	135
17.18[169] “GeomContour” “GeomContourFilled”	135
17.18[171] “GeomCrossbar” “GeomCurve”	135
17.18[173] “GeomCustomAnn” “GeomDensity”	135
17.18[175] “GeomDensity2d” “GeomDensity2dFilled”	135
17.18[177] “GeomDotplot” “GeomErrorbar”	135
17.18[179] “GeomErrorbarh” “GeomFunction”	135
17.18[181] “GeomHex” “GeomHline”	135

17.18[183] “GeomLabel” “GeomLine”	135
17.19[185] “GeomLinerange” “GeomLogticks”	135
17.19[187] “GeomMap” “GeomPath”	135
17.19[189] “GeomPoint” “GeomPointrange”	135
17.19[191] “GeomPolygon” “GeomQuantile”	135
17.19[193] “GeomRaster” “GeomRasterAnn”	135
17.19[195] “GeomRect” “GeomRibbon”	135
17.19[197] “GeomRug” “GeomSegment”	135
17.19[199] “GeomSf” “GeomSmooth”	135
17.19[201] “GeomSpoke” “GeomStep”	135
17.19[203] “GeomText” “GeomTile”	135
17.20[205] “GeomViolin” “GeomVline”	135
17.20[207] “get_alt_text” “get_element_tree”	135
17.20[209] “get_geom_defaults” “get_guide_data”	135
17.20[211] “get_labs” “gg_dep”	135
17.20[213] “ggplot” “ggplot_add”	135
17.20[215] “ggplot_build” “ggplot_gtable”	135
17.20[217] “ggplotGrob” “ggproto”	135
17.20[219] “ggproto_parent” “ggsave”	135
17.20[221] “ggttitle” “Guide”	135
17.20[223] “guide_axis” “guide_axis_logticks”	135
17.21[225] “guide_axis_stack” “guide_axis_theta”	135
17.21[227] “guide_bins” “guide_colorbar”	135
17.21[229] “guide_colorsteps” “guide_colourbar”	135
17.21[231] “guide_coloursteps” “guide_custom”	135
17.21[233] “guide_gengrob” “guide_geom”	135
17.21[235] “guide_legend” “guide_merge”	135
17.21[237] “guide_none” “guide_train”	135
17.21[239] “guide_transform” “GuideAxis”	135
17.21[241] “GuideAxisLogticks” “GuideAxisStack”	135

17.21[243] “GuideAxisTheta” “GuideBins”	135
17.22[245] “GuideColourbar” “GuideColoursteps”	135
17.22[247] “GuideCustom” “GuideLegend”	135
17.22[249] “GuideNone” “GuideOld”	135
17.22[251] “guides” “has_flipped_aes”	135
17.22[253] “is_coord” “is_facet”	135
17.22[255] “is_geom” “is_ggplot”	135
17.22[257] “is_ggproto” “is_guide”	135
17.22[259] “is_guides” “is_layer”	135
17.22[261] “is_mapping” “is_margin”	135
17.22[263] “is_position” “is_scale”	135
17.23[265] “is_stat” “is_theme”	135
17.23[267] “is_theme_element” “is.Coord”	135
17.23[269] “is.facet” “is.ggplot”	135
17.23[271] “is.ggproto” “is.theme”	135
17.23[273] “label_both” “label_bquote”	135
17.23[275] “label_context” “label_parsed”	135
17.23[277] “label_value” “label_wrap_gen”	135
17.23[279] “labeller” “labs”	135
17.23[281] “last_plot” “layer”	135
17.23[283] “layer_data” “layer_grob”	135
17.24[285] “layer_scales” “layer_sf”	135
17.24[287] “Layout” “lims”	135
17.24[289] “luv_colours” “map_data”	135
17.24[291] “margin” “max_height”	135
17.24[293] “max_width” “mean_cl_boot”	135
17.24[295] “mean_cl_normal” “mean sdl”	135
17.24[297] “mean_se” “median_hilow”	135
17.24[299] “merge_element” “midwest”	135
17.24[301] “mpg” “msleep”	135

17.24[303] “new_guide” “old_guide”	135
17.25[305] “panel_cols” “panel_rows”	135
17.25[307] “pattern_alpha” “Position”	135
17.25[309] “position_dodge” “position_dodge2”	135
17.25[311] “position_fill” “position_identity”	135
17.25[313] “position_jitter” “position_jitterdodge”	135
17.25[315] “position_nudge” “position_stack”	135
17.25[317] “PositionDodge” “PositionDodge2”	135
17.25[319] “PositionFill” “PositionIdentity”	135
17.25[321] “PositionJitter” “PositionJitterdodge”	135
17.25[323] “PositionNudge” “PositionStack”	135
17.26[325] “presidential” “qplot”	135
17.26[327] “quickplot” “quo”	135
17.26[329] “quo_name” “quos”	135
17.26[331] “register_theme_elements” “rel”	135
17.26[333] “remove_missing” “render_axes”	135
17.26[335] “render_strips” “reset_theme_settings”	135
17.26[337] “resolution” “Scale”	135
17.26[339] “scale_alpha” “scale_alpha_binned”	135
17.26[341] “scale_alpha_continuous” “scale_alpha_date”	135
17.26[343] “scale_alpha_datetime” “scale_alpha_discrete”	135
17.27[345] “scale_alpha_identity” “scale_alpha_manual”	135
17.27[347] “scale_alpha_ordinal” “scale_color_binned”	135
17.27[349] “scale_color_brewer” “scale_color_continuous”	135
17.27[351] “scale_color_date” “scale_color_datetime”	135
17.27[353] “scale_color_discrete” “scale_color_distiller”	135
17.27[355] “scale_color_fermenter” “scale_color_gradient”	135
17.27[357] “scale_color_gradient2” “scale_color_gradientn”	135
17.27[359] “scale_color_grey” “scale_color_hue”	135
17.27[361] “scale_color_identity” “scale_color_manual”	135

17.27[363] “scale_color_ordinal” “scale_color_steps”	135
17.28[365] “scale_color_steps2” “scale_color_stepsn”	135
17.28[367] “scale_color_viridis_b” “scale_color_viridis_c”	135
17.28[369] “scale_color_viridis_d” “scale_colour_binned”	135
17.28[371] “scale_colour_brewer” “scale_colour_continuous”	135
17.28[373] “scale_colour_date” “scale_colour_datetime”	135
17.28[375] “scale_colour_discrete” “scale_colour_distiller”	135
17.28[377] “scale_colour_fermenter” “scale_colour_gradient”	135
17.28[379] “scale_colour_gradient2” “scale_colour_gradients”	135
17.28[381] “scale_colour_grey” “scale_colour_hue”	135
17.28[383] “scale_colour_identity” “scale_colour_manual”	135
17.29[385] “scale_colour_ordinal” “scale_colour_steps”	135
17.29[387] “scale_colour_steps2” “scale_colour_stepsn”	135
17.29[389] “scale_colour_viridis_b” “scale_colour_viridis_c”	135
17.29[391] “scale_colour_viridis_d” “scale_continuous_identity”	135
17.29[393] “scale_discrete_identity” “scale_discrete_manual”	135
17.29[395] “scale_fill_binned” “scale_fill_brewer”	135
17.29[397] “scale_fill_continuous” “scale_fill_date”	135
17.29[399] “scale_fill_datetime” “scale_fill_discrete”	135
17.29[401] “scale_fill_distiller” “scale_fill_fermenter”	135
17.29[403] “scale_fill_gradient” “scale_fill_gradient2”	135
17.30[405] “scale_fill_gradientn” “scale_fill_grey”	135
17.30[407] “scale_fill_hue” “scale_fill_identity”	135
17.30[409] “scale_fill_manual” “scale_fill_ordinal”	135
17.30[411] “scale_fill_steps” “scale_fill_steps2”	135
17.30[413] “scale_fill_stepsn” “scale_fill_viridis_b”	135
17.30[415] “scale_fill_viridis_c” “scale_fill_viridis_d”	135
17.30[417] “scale_linetype” “scale_linetype_binned”	135
17.30[419] “scale_linetype_continuous” “scale_linetype_discrete”	135
17.30[421] “scale_linetype_identity” “scale_linetype_manual”	135

17.30[423] “scale_linewidth” “scale_linewidth_binned”	135
17.31[425] “scale_linewidth_continuous” “scale_linewidth_date”	135
17.31[427] “scale_linewidth_datetime” “scale_linewidth_discrete”	135
17.31[429] “scale_linewidth_identity” “scale_linewidth_manual”	135
17.31[431] “scale_linewidth_ordinal” “scale_radius”	135
17.31[433] “scale_shape” “scale_shape_binned”	135
17.31[435] “scale_shape_continuous” “scale_shape_discrete”	135
17.31[437] “scale_shape_identity” “scale_shape_manual”	135
17.31[439] “scale_shape_ordinal” “scale_size”	135
17.31[441] “scale_size_area” “scale_size_binned”	135
17.31[443] “scale_size_binned_area” “scale_size_continuous”	135
17.32[445] “scale_size_date” “scale_size_datetime”	135
17.32[447] “scale_size_discrete” “scale_size_identity”	135
17.32[449] “scale_size_manual” “scale_size_ordinal”	135
17.32[451] “scale_type” “scale_x_binned”	135
17.32[453] “scale_x_continuous” “scale_x_date”	135
17.32[455] “scale_x_datetime” “scale_x_discrete”	135
17.32[457] “scale_x_log10” “scale_x_reverse”	135
17.32[459] “scale_x_sqrt” “scale_x_time”	135
17.32[461] “scale_y_binned” “scale_y_continuous”	135
17.32[463] “scale_y_date” “scale_y_datetime”	135
17.33[465] “scale_y_discrete” “scale_y_log10”	135
17.33[467] “scale_y_reverse” “scale_y_sqrt”	135
17.33[469] “scale_y_time” “ScaleBinned”	135
17.33[471] “ScaleBinnedPosition” “ScaleContinuous”	135
17.33[473] “ScaleContinuousDate” “ScaleContinuousDatetime”	135
17.33[475] “ScaleContinuousIdentity” “ScaleContinuousPosition”	135
17.33[477] “ScaleDiscrete” “ScaleDiscreteIdentity”	135
17.33[479] “ScaleDiscretePosition” “seals”	135
17.33[481] “sec_axis” “set_last_plot”	135

17.33[483] “sf_transform_xy” “should_stop”	135
17.34[485] “stage” “standardise_aes_names”	135
17.34[487] “stat” “Stat”	135
17.34[489] “stat_align” “stat_bin”	135
17.34[491] “stat_bin_2d” “stat_bin_hex”	135
17.34[493] “stat_bin2d” “stat_binhex”	135
17.34[495] “stat_boxplot” “stat_contour”	135
17.34[497] “stat_contour_filled” “stat_count”	135
17.34[499] “stat_density” “stat_density_2d”	135
17.34[501] “stat_density_2d_filled” “stat_density2d”	135
17.34[503] “stat_density2d_filled” “stat_ecdf”	135
17.35[505] “stat_ellipse” “stat_function”	135
17.35[507] “stat_identity” “stat_qq”	135
17.35[509] “stat_qq_line” “stat_quantile”	135
17.35[511] “stat_sf” “stat_sf_coordinates”	135
17.35[513] “stat_smooth” “stat_spoke”	135
17.35[515] “stat_sum” “stat_summary”	135
17.35[517] “stat_summary_2d” “stat_summary_bin”	135
17.35[519] “stat_summary_hex” “stat_summary2d”	135
17.35[521] “stat_unique” “stat_ydensity”	135
17.35[523] “StatAlign” “StatBin”	135
17.36[525] “StatBin2d” “StatBindot”	135
17.36[527] “StatBinhex” “StatBoxplot”	135
17.36[529] “StatContour” “StatContourFilled”	135
17.36[531] “StatCount” “StatDensity”	135
17.36[533] “StatDensity2d” “StatDensity2dFilled”	135
17.36[535] “StatEcdf” “StatEllipse”	135
17.36[537] “StatFunction” “StatIdentity”	135
17.36[539] “StatQq” “StatQqLine”	135
17.36[541] “StatQuantile” “StatSf”	135

17.36[543] “StatSfCoordinates” “StatSmooth”	135
17.37[545] “StatSum” “StatSummary”	135
17.37[547] “StatSummary2d” “StatSummaryBin”	135
17.37[549] “StatSummaryHex” “StatUnique”	135
17.37[551] “StatYdensity” “summarise_coord”	135
17.37[553] “summarise_layers” “summarise_layout”	135
17.37[555] “sym” “syms”	135
17.37[557] “theme” “theme_bw”	135
17.37[559] “theme_classic” “theme_dark”	135
17.37[561] “theme_get” “theme_gray”	135
17.37[563] “theme_grey” “theme_light”	135
17.38[565] “theme_linedraw” “theme_minimal”	135
17.38[567] “theme_replace” “theme_set”	135
17.38[569] “theme_test” “theme_update”	135
17.38[571] “theme_void” “transform_position”	135
17.38[573] “translate_shape_string” “txhousing”	135
17.38[575] “unit” “update_geom_defaults”	135
17.38[577] “update_labels” “update_stat_defaults”	135
17.38[579] “vars” “waiver”	135
17.38[581] “wrap_dims” “xlab”	135
17.38[583] “xlim” “ylab”	135
17.39[585] “ylim” “zeroGrob”	135
17.39⑥ Subsetting: vectors and factors	136
17.39② 15 22 45 52 73 81	136
17.39③ 73	136
17.39④ 15 22 45 52 81	136
17.39⑤ 45 73 81	136
17.39⑥ 45 73 81	137
17.39⑦ 15 22 45 52	137
17.39⑧ 15 22 45 52 73 81	137

17.39 ¶] FALSE FALSE FALSE TRUE TRUE TRUE	137
17.40 ¶] 52 73 81	138
17.40 ¶] TRUE FALSE FALSE TRUE TRUE TRUE	138
17.40 ¶] 15 22 45 52 73 81	138
17.40 ¶] 15 52 73 81	138
17.40 ¶] TRUE FALSE FALSE TRUE TRUE TRUE	138
17.40 ¶] 15 52 73 81	138
17.40 ¶] 1 4 5 6	138
17.40 ¶] 15 52 73 81	139
17.40 ¶] 15 52 73 81	139
17.40 ¶] low high medium high low medium high	139
17.41 ¶ Levels: high low medium	139
17.41Factor w/ 3 levels “high”,“low”,“medium”: 2 1 3 1 2 3 1	139
17.41 ¶] high high high	139
17.41 ¶ Levels: high low medium	139
17.41 ¶] high high high	140
17.41 ¶ Levels: high low medium	140
17.41 ¶] high high high	140
17.41 ¶ Levels: high	140
17.41 ¶] “high” “high” “high”	140
17.41 ¶] low high medium high low medium high	140
17.42 ¶ Levels: high low medium	140
17.42Factor w/ 3 levels “low”,“medium”,..: 1 3 2 3 1 2 3	141
17.42 ¶ 07 Reading and data inspection	141
17.42 ¶ genotype celltype replicate	142
17.42 ¶ sample1 Wt typeA 1	142
17.42 ¶ sample2 Wt typeA 2	142
17.42 ¶ sample3 Wt typeA 3	142
17.42 ¶ sample4 KO typeA 1	142
17.42 ¶ sample5 KO typeA 2	142

17.42	sample6 KO typeA 3	142
17.43	sample7 Wt typeB 1	142
17.43	sample8 Wt typeB 2	142
17.43	sample9 Wt typeB 3	142
17.43	sample10 KO typeB 1	142
17.43	sample11 KO typeB 2	142
17.43	sample12 KO typeB 3	142
17.43	genotype celltype replicate	143
17.43	sample1 Wt typeA 1	143
17.43	sample2 Wt typeA 2	143
17.43	sample3 Wt typeA 3	143
17.44	sample4 KO typeA 1	143
17.44	sample5 KO typeA 2	143
17.44	sample6 KO typeA 3	143
17.44	data.frame': 12 obs. of 3 variables:	143
17.44\$	genotype : chr "Wt" "Wt" "Wt" "KO"	143
17.44\$	celltype : chr "typeA" "typeA" "typeA" "typeA"	143
17.44\$	replicate: int 1 2 3 1 2 3 1 2 3 1	143
17.44[1]	12 3	144
17.44[1]	12	144
17.44[1]	3	144
17.45[1]	"data.frame"	144
17.45[1]	"genotype" "celltype" "replicate"	144
17.45	08 Data frames and matrices	145
17.45[1]	"Wt"	146
17.45\$	genotype celltype replicate	146
17.45	sample3 Wt typeA 3	146
17.45[1]	1 2 3 1 2 3 1 2 3 1 2 3	146
17.45\$	replicate	147
17.45	sample1 1	147

17.45	sample2	2	147
17.46	sample3	3	147
17.46	sample4	1	147
17.46	sample5	2	147
17.46	sample6	3	147
17.46	sample7	1	147
17.46	sample8	2	147
17.46	sample9	3	147
17.46	sample10	1	147
17.46	sample11	2	147
17.46	sample12	3	147
17.47	genotype	celltype	148
17.47	sample1	Wt typeA	148
17.47	sample2	Wt typeA	148
17.47	sample3	Wt typeA	148
17.47	sample4	KO typeA	148
17.47	sample5	KO typeA	148
17.47	sample6	KO typeA	148
17.47	sample7	Wt typeB	148
17.47	sample8	Wt typeB	148
17.47	sample9	Wt typeB	148
17.48	sample10	KO typeB	148
17.48	sample11	KO typeB	148
17.48	sample12	KO typeB	148
17.48	genotype	celltype replicate	149
17.48	sample1	Wt typeA 1	149
17.48	sample3	Wt typeA 3	149
17.48	sample6	KO typeA 3	149
17.48	[1]	“typeA” “typeA” “typeA”	149
17.48	[1]	“genotype” “celltype” “replicate”	149

17.48[1] "sample1" "sample2" "sample3" "sample4" "sample5" "sample6"	149
17.49[7] "sample7" "sample8" "sample9" "sample10" "sample11" "sample12"	149
17.49[11] "Wt" "Wt" "Wt" "KO" "KO" "KO" "Wt" "Wt" "Wt" "KO" "KO" "KO"	149
17.49[21] "Wt" "Wt" "Wt" "KO" "KO"	150
17.49[31] "genotype" "celltype" "replicate"	150
17.49[41] "typeA" "typeA" "typeA" "typeA" "typeA" "typeB" "typeB" "typeB"	150
17.49[510] "typeB" "typeB" "typeB"	150
17.496	151
17.497typeA typeB	151
17.498 6	151
17.499[1] TRUE TRUE TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE	151
17.500genotype celltype replicate	151
17.501sample1 Wt typeA 1	151
17.502sample2 Wt typeA 2	151
17.503sample3 Wt typeA 3	151
17.504sample4 KO typeA 1	151
17.505sample5 KO typeA 2	151
17.506sample6 KO typeA 3	151
17.507genotype celltype replicate	152
17.508sample1 Wt typeA 1	152
17.509sample2 Wt typeA 2	152
17.510sample3 Wt typeA 3	152
17.511sample4 KO typeA 1	152
17.512sample5 KO typeA 2	152
17.513sample6 KO typeA 3	152
17.514genotype celltype replicate	152
17.515sample1 Wt typeA 1	152
17.516sample2 Wt typeA 2	152
17.517sample3 Wt typeA 3	152
17.518sample4 KO typeA 1	152

17.519ample5 KO typeA 2	152
17.520ample6 KO typeA 3	152
17.521	153
17.522 2 3	153
17.523 4 4	153
17.524[1] 2 3 5 6 8 9 11 12	153
17.525genotype celltype replicate	153
17.526ample2 Wt typeA 2	153
17.527ample3 Wt typeA 3	153
17.528ample5 KO typeA 2	153
17.529ample6 KO typeA 3	153
17.530ample8 Wt typeB 2	153
17.531ample9 Wt typeB 3	153
17.532ample11 KO typeB 2	153
17.533ample12 KO typeB 3	153
17.534Logical operators for matching	154
17.535ample2 sample5 sample7 sample8 sample9 sample4	156
17.536ENSMUSG000000000001 19.265000 23.7222000 2.611610 5.8495400 6.5126300 24.076700	156
17.537ENSMUSG000000000003 0.000000 0.0000000 0.000000 0.0000000 0.0000000 0.000000	156
17.538ENSMUSG000000000028 1.032290 0.8269540 1.134410 0.6987540 0.9251170 0.827891	156
17.539ENSMUSG000000000031 0.000000 0.0000000 0.000000 0.0298449 0.0597726 0.000000	156
17.540ENSMUSG000000000037 0.056033 0.0473238 0.000000 0.0685938 0.0494147 0.180883	156
17.541ENSMUSG000000000049 0.258134 1.0730200 0.252342 0.2970320 0.2082800 2.191720	156
17.542ample6 sample12 sample3 sample11 sample10	156
17.543ENSMUSG000000000001 20.8198000 26.9158000 20.889500 24.0465000 24.198100	156
17.544ENSMUSG000000000003 0.0000000 0.0000000 0.000000 0.0000000 0.000000	156
17.545ENSMUSG000000000028 1.1686300 0.6735630 0.892183 0.9753270 1.045920	156
17.546ENSMUSG000000000031 0.0511932 0.0204382 0.000000 0.0000000 0.000000	156
17.547ENSMUSG000000000037 0.1438840 0.0662324 0.146196 0.0206405 0.017004	156
17.548ENSMUSG000000000049 1.6853800 0.1161970 0.421286 0.0634322 0.369550	156

17.54	sample1	156	
17.55	ENSMUSG000000000001	19.7848000	156
17.55	ENSMUSG000000000003	0.0000000	156
17.55	ENSMUSG000000000028	0.9377920	156
17.55	ENSMUSG000000000031	0.0359631	156
17.55	ENSMUSG000000000037	0.1514170	156
17.55	ENSMUSG000000000049	0.2567330	156
17.55	[1] "sample2" "sample5" "sample7" "sample8" "sample9" "sample4"	157	
17.55	[7] "sample6" "sample12" "sample3" "sample11" "sample10" "sample1"	157	
17.55	[8] "sample1" "sample2" "sample3" "sample4" "sample5" "sample6"	157	
17.55	[7] "sample7" "sample8" "sample9" "sample10" "sample11" "sample12"	157	
17.56	[1] 12	157	
17.56	[1] 12	157	
17.56	[1] FALSE FALSE FALSE FALSE FALSE FALSE	158	
17.56	[1] TRUE FALSE TRUE FALSE FALSE FALSE	158	
17.56	[1] TRUE FALSE TRUE FALSE FALSE FALSE	158	
17.56	[1] 1 5	158	
17.56	[1] TRUE	158	
17.56	[1] FALSE	158	
17.56	[1] TRUE TRUE TRUE TRUE TRUE	159	
17.56	[1] FALSE FALSE TRUE FALSE FALSE	159	
17.57	[1] FALSE	159	
17.57	[1] TRUE	159	
17.57	[1] TRUE	159	
17.57	[1] FALSE	160	
17.57	[1] FALSE	160	
17.57	\$0 Reordering to match datasets	160	
17.57	[1] "A" "B" "C" "D" "E"	161	
17.57	[1] "B" "D" "E" "A" "C"	161	
17.57	[1] 4 1 5 2 3	161	

17.57 [¶]] "A" "B" "C" "D" "E"	161
17.58 [¶]] TRUE TRUE TRUE TRUE TRUE	161
17.58 ^{¶1}] 3 2 NA 1 NA	162
17.58 ^{¶2}] "A" "B" NA "D" NA	162
17.58 ^{¶3}] 3 2 0 1 0	162
17.58 ^{¶4}] "A" "B" "D"	162
17.58 ^{¶5}] "sample1" "sample2" "sample3" "sample4" "sample5" "sample6"	162
17.58 ^{¶6}] "sample7" "sample8" "sample9" "sample10" "sample11" "sample12"	162
17.58 ^{¶7}] "sample2" "sample5" "sample7" "sample8" "sample9" "sample4"	162
17.58 ^{¶8}] "sample6" "sample12" "sample3" "sample11" "sample10" "sample1"	162
17.58 ^{¶9}] 12 1 9 6 2 7 3 4 5 11 10 8	163
17.59 [¶] sample1 sample2 sample3 sample4 sample5	164
17.59 [¶] ENSMUSG000000000001 19.7848000 19.265000 20.889500 24.076700 23.7222000	164
17.59 [¶] ENSMUSG000000000003 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000	164
17.59 [¶] ENSMUSG000000000028 0.9377920 1.032290 0.892183 0.827891 0.8269540	164
17.59 [¶] ENSMUSG000000000031 0.0359631 0.0000000 0.0000000 0.0000000 0.0000000	164
17.59 [¶] ENSMUSG000000000037 0.1514170 0.056033 0.146196 0.180883 0.0473238	164
17.59 [¶] ENSMUSG000000000049 0.2567330 0.258134 0.421286 2.191720 1.0730200	164
17.59 ^{¶1}] TRUE	164
17.59 ^{¶1} Plotting and data visualization	165
17.59 ^{¶1}] 10.2661	165
17.60 [¶] genotype celltype replicate samplemeans age_in_days	166
17.60 [¶] sample1 Wt typeA 1 10.266102 40	166
17.60 [¶] sample2 Wt typeA 2 10.849759 32	166
17.60 [¶] sample3 Wt typeA 3 9.452517 38	166
17.60 [¶] sample4 KO typeA 1 15.833872 35	166
17.60 [¶] sample5 KO typeA 2 15.590184 41	166
17.60 [¶] sample6 KO typeA 3 15.551529 32	166
17.60 ^{¶2}] Data visualization with ggplot2	166
17.60 ^{¶8} genotype celltype replicate samplemeans age_in_days	167

17.60	sample1 Wt typeA 1 10.266102 40	167
17.61	sample2 Wt typeA 2 10.849759 32	167
17.61	sample3 Wt typeA 3 9.452517 38	167
17.61	sample4 KO typeA 1 15.833872 35	167
17.61	sample5 KO typeA 2 15.590184 41	167
17.61	sample6 KO typeA 3 15.551529 32	167
17.61	\$3 Boxplot exercise points +16	174
17.61	\$4 Saving data and plots to file	176
17.61	pdf	177
17.61	8	177
17.61	5 Finding help	177
17.62	06 Tidyverse data wrangling	178
17.62	[1] 9.110434	178
17.62	[2] 9.11	178
17.62	[3] 9.11	178
17.62	4 query.number significant p.value term.size query.size overlap.size recall	180
17.62	5 1 TRUE 0.00434 111 5850 52 0.009	180
17.62	0 1 TRUE 0.00330 110 5850 52 0.009	180
17.62	3 1 TRUE 0.02970 39 5850 21 0.004	180
17.62	8 precision term.id domain subgraph.number	180
17.62	9 0.468 GO:0032606 BP 237	180
17.63	0 0.473 GO:0032479 BP 237	180
17.63	3 0.538 GO:0032480 BP 237	180
17.63	2 term.name	180
17.63	3 type I interferon production	180
17.63	2 regulation of type I interferon production	180
17.63	3 negative regulation of type I interferon production	180
17.63	4 [1] “query.number” “significant” “p.value” “term.size”	180
17.63	5 [5] “query.size” “overlap.size” “recall” “precision”	180
17.63	9 [9] “term.id” “domain” “subgraph.number” “term.name”	180

17.63[3] “relative.depth” “intersection”	180
17.64[1] “query.number” “significant” “p.value” “term.size”	181
17.64[5] “query.size” “overlap.size” “recall” “precision”	181
17.64[9] “term.id” “domain” “subgraph.number” “term.name”	181
17.64[13] “relative.depth” “intersection”	181
17.64[14] query.number significant p.value term.size query.size overlap.size recall	182
17.64[5] 1 TRUE 0.00434 111 5850 52 0.009	182
17.64[0] 1 TRUE 0.00330 110 5850 52 0.009	182
17.64[3] 1 TRUE 0.02970 39 5850 21 0.004	182
17.64[8] precision term.id domain subgraph.number	182
17.64[9] 0.468 GO:0032606 BP 237	182
17.65[0] 0.473 GO:0032479 BP 237	182
17.65[3] 0.538 GO:0032480 BP 237	182
17.65[2] term.name	182
17.65[3] type I interferon production	182
17.65[2] regulation of type I interferon production	182
17.65[3] negative regulation of type I interferon production	182
17.65[6] [1] “query.number” “significant” “p.value” “term.size”	183
17.65[5] [5] “query.size” “overlap.size” “recall” “precision”	183
17.65[8] [9] “term.id” “domain” “subgraph.number” “term.name”	183
17.65[9] [13] “relative.depth” “intersection”	183
17.66[0] term.id term.name p.value query.size	184
17.66[1] GO:0090672 telomerase RNA localization 2.41e-02 5850	184
17.66[2] GO:0090670 RNA localization to Cajal body 2.41e-02 5850	184
17.66[3] GO:0090671 telomerase RNA localization to Cajal body 2.41e-02 5850	184
17.66[4] GO:0071702 organic substance transport 7.90e-11 5850	184
17.66[5] GO:0015931 nucleobase-containing compound transport 4.43e-05 5850	184
17.66[6] GO:0050657 nucleic acid transport 3.67e-06 5850	184
17.66[7] term.size overlap.size	184
17.66[8] 16 11	184

17.66 ⁹	16	11	184
17.67 ⁰	16	11	184
17.67 ¹	2629	973	184
17.67 ²	200	93	184
17.67 ³	166	83	184
17.67 ^[4]	668	7	185

Welcome to Omics Science!

The lab module includes four sections:

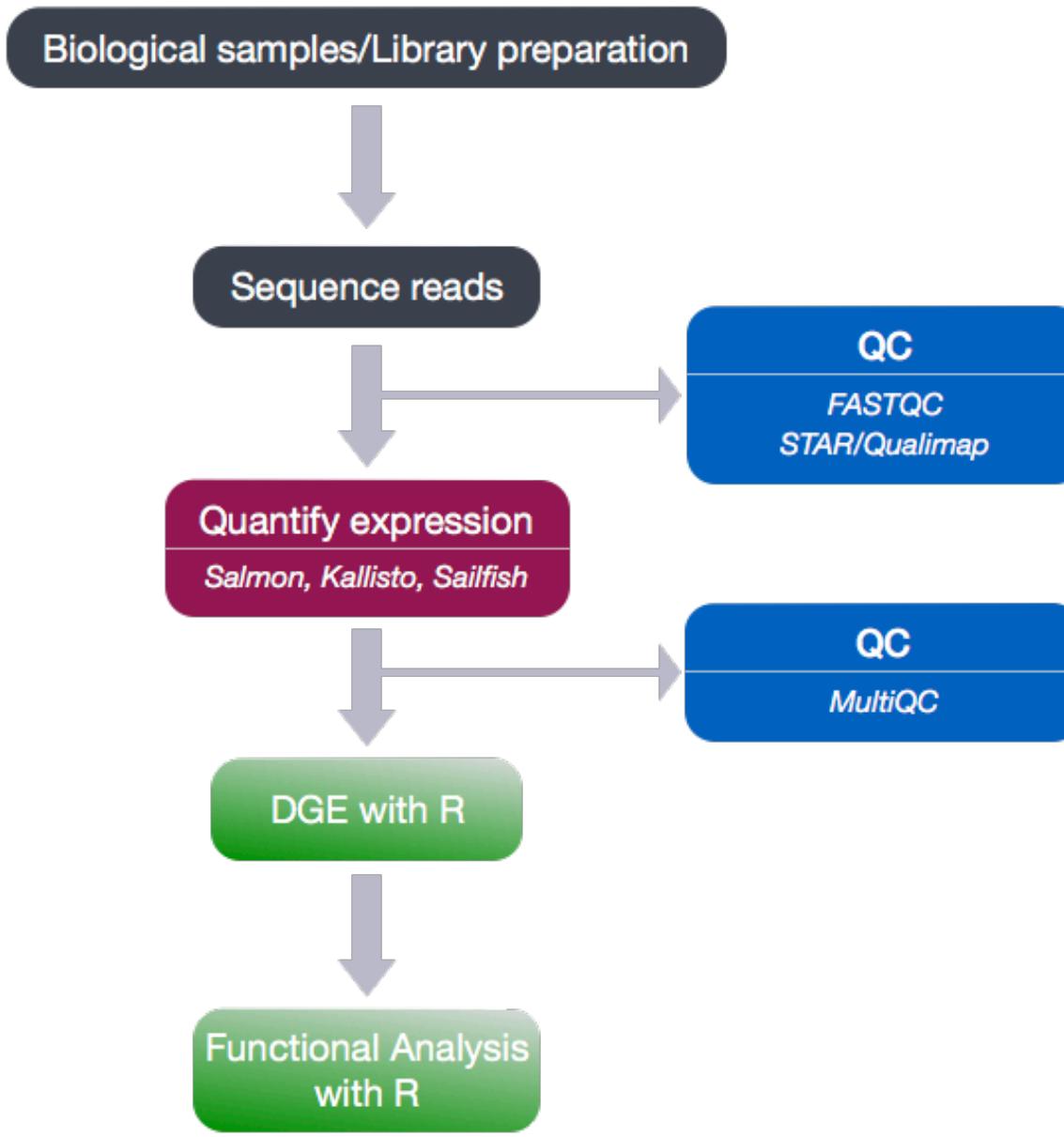
- 1. Introduction to R and RStudio. Bookdown site: [2025_IntroR_and_RStudio](#) Github site: [2025_IntroR_and_RStudio](#)
- 2. Introduction to Differential Gene Expression (DGE) analysis. Bookdown site: [2025_RNA-Seq-Analysis](#). Github site: [2025_RNA-Seq-Analysis](#).
- 3. Differential Gene Expression (DGE) continued. Bookdown site: [2025_RNA-Seq-Analysis](#). Github site: [2025_RNA-Seq-Analysis](#).
- 4. Data Visualization in R. Bookdown site: [2025_r4ds-ggplot2](#). Github site: [2025_r4ds-ggplot2](#)
- 5. Variant-calling and annotation pipeline: **Marjan Barazandeh**

What we will cover:

This course will cover several of the statistical concepts and data analytic skills needed to succeed in data-driven life science research as it relates to genomics and the omic sciences. For the bulk of the course we cover topics related to genomics and high-dimensional data. Here we cover experimental techniques used in genomics including RNA-seq and variant analysis.

We start with an introduction to R, including data structures, data wrangling and plotting methods. We then walk you through an end-to-end gene-level RNA-seq differential expression workflow using various R packages. We will start with the count matrix, perform exploratory data analysis for quality assessment and to explore the relationship between samples, perform differential expression analysis, and visually explore the results prior to performing downstream functional analysis.

To determine the expression levels of genes, the full RNA-seq workflow follows the steps detailed in the image below. All steps are performed on the command line (Linux/Unix) through the generation of the read counts per gene. The differential expression analysis and downstream functional analysis are generally performed in R using R packages specifically designed for the complex statistical analyses required to determine whether genes are differentially expressed.



{ width=400 }

We will cover the concepts of distance and dimension reduction. We will introduce Principal Component Analysis for dimension reduction. Principal Component Analysis (PCA) is a technique used to emphasize variation and bring out strong patterns in a dataset. Once we learn this, we will be ready to cover hierarchical clustering and other methods of visualization.

We will next cover variant analysis; from FASTQ files to mapping genome sequencing data to a reference genome to produce high-quality variant calls that can be used in downstream analyses. We will use a pipeline employing the Genome Analysis Toolkit (GATK) to perform variant calling that is based on the best practices for variant discovery analysis outlined by the Broad Institute (MIT). The result will be the identification of genomic variants, such as single nucleotide polymorphisms (SNPs) and DNA insertions and deletions (indels).

Finally, we will cover functional analysis of high-throughput data. The output of RNA-Seq differential expression analysis is a list of significant differentially expressed genes (DEGs). To gain greater biological insight on the differentially expressed genes there are various analyses that can be done:

- determine whether there is enrichment of known biological functions, interactions, or pathways

- identify genes' involvement in novel pathways or networks by grouping genes together based on similar trends
- use global changes in gene expression by visualizing all genes being significantly up- or down-regulated in the context of external interaction data

How is this course different?

While statistics textbooks focus on mathematics, this book focuses on using a computer to perform data analysis by stating a practical data-related challenge. This book also includes the computer code that provides a solution to the problem and helps illustrate the concepts behind the solution. By running the code yourself, and seeing data generation and analysis happen live, you will get a better intuition for the concepts, the mathematics, and the theory.

Throughout the course we show the R code that performs genomic analysis. All sections of this book are reproducible; comprised of *R markdown* documents that include the R code necessary to produce all figures, tables and results.

Let's get started with the first module, Introduction to R and RStudio.

Chapter 1

Getting Started with R and RStudio

1.1 A Beginner’s Guide for First-Year Graduate Students

1.2 What You’re Installing and Why

Before we start, here’s what each tool does:

- **R**: The programming language that does the actual data analysis
- **RStudio**: A user-friendly interface that makes R easier to use (like Microsoft Word for documents)
- **Git**: Software that helps you download and sync course materials
- **Additional tools**: Help R work with different types of data and create documents

Important: Install everything in the order listed below. Each step builds on the previous one.

1.3 Step 1: Install R

R is the programming language. Install this first!

1.3.1 For Windows:

1. Go to R Project website
2. Click “base” then “Download R for Windows”
3. Run the downloaded file and follow the installation prompts

1.3.2 For macOS:

These instructions work for ALL Macs (Intel and Apple Silicon):

1. Go to R Project website

2. Download the **Intel (x86_64)** version (yes, even for newer Macs with M1/M2/M3 chips)
3. Run the downloaded .pkg file and follow the installation prompts

1.4 Step 2: Install Additional Tools (macOS Only)

Windows users can skip to Step 3. Mac users need these tools for R to work properly:

1.4.1 Install Required System Tools:

Open **Terminal** (found in Applications → Utilities) and paste each command one at a time:

```
xcode-select --install
```

Click “Install” when prompted and wait for it to finish (this may take 10-15 minutes).

1.4.2 Install Graphics Support:

1. Go to XQuartz.org
2. Download and install XQuartz
3. **Log out and log back in** to your Mac after installation

1.4.3 Install Fortran Compiler:

1. Go to R Tools for macOS
2. Download **gfortran-12.2-universal.pkg**
3. Install it by double-clicking the downloaded file

1.4.4 Add Fortran to Your System Path:

In Terminal, paste these commands:

```
echo 'export PATH="/opt/gfortran/bin:/usr/local/bin:$PATH"' >> ~/.zshrc
echo 'export PATH="/opt/gfortran/bin:/usr/local/bin:$PATH"' >> ~/.bash_profile
```

1.5 Step 3: Install Git

Git helps you download and sync course materials.

1.5.1 For Windows:

1. Go to git-scm.com
2. Download Git for Windows
3. Run the installer using all default settings (just keep clicking “Next”)

1.5.2 For macOS:

If you completed Step 2, Git is already installed! To verify, open Terminal and type:

```
git --version
```

You should see version information.

1.5.3 Configure Git (All Users):

Open **Terminal** (Mac) or **Git Bash** (Windows) and run these commands with your information:

```
git config --global user.name "YourFullName"
git config --global user.email "your.email@university.edu"
```

1.6 Step 4: Install RStudio

RStudio is the user-friendly interface for R.

1. Go to RStudio.com
2. Download **RStudio Desktop** (the free version)
3. Install it like any other program

1.6.1 For Mac Users with Apple Silicon (M1/M2/M3):

After installing RStudio:

1. In Finder, go to **Applications**
2. Right-click on **RStudio.app**
3. Select **Get Info**
4. Check the box “**Open using Rosetta**”

1.7 Step 5: Test Your Installation

Let's make sure everything works:

1.7.1 Test RStudio and R:

1. **Open RStudio** (not R - always use RStudio)
2. You should see 4 panels:
 - **Console** (bottom left) - this is where you type R commands
 - **Environment** (top right) - shows your data
 - **Files/Plots** (bottom right) - shows files and graphs

- **Script** (top left) - for writing longer code
3. **Click in the Console** (the bottom left panel where you see >)
 4. Type this and press Enter:

```
2 + 2
```

You should see [1] 4

1.7.2 Test Git Connection:

1. In RStudio, go to **File → New Project → Version Control**
2. You should see “**Git**” as an option
3. If you see it, great! If not, restart RStudio and try again

1.8 Step 6: Download Course Materials

Now let's get the main course files:

1. In RStudio, go to **File → New Project → Version Control → Git**
2. In the “Repository URL” field, paste:

```
https://github.com/gurinina/2025_IntroR_and_RStudio
```

3. Choose where to save the project on your computer
4. Click “**Create Project**”

RStudio will download all course materials. This may take a few minutes.

1.8.1 Additional Repositories:

As the course progresses, you may need to clone additional repositories for specific modules. Your instructor will provide those URLs when needed.

Tips: - To **update** materials later in the term, open the project and click the **Pull** button in the **Git** tab - To **switch** between projects later: RStudio → *File* → *Open Project...* and select the corresponding **.Rproj** file

1.8.2 Understanding the Git Pane:

When you open your course project in RStudio, you should see a tab labeled **Git** in the upper-right panel (next to *Environment* and *History*). This is where you can:

- **See changes** you've made to files (they'll appear in the list)
- **Revert changes** if you made an edit you don't want to keep

- **Pull** to update your local copy with the newest course materials from GitHub
- **Push** to send your own changes to GitHub (*not applicable in this course — you won't be pushing*)

For this course, you will mainly use the **Pull** button (blue down-arrow icon) to update your files when the instructor posts new material.

1.8.3 IMPORTANT: Git Best Practices for Students

To avoid problems when updating course materials, follow these rules:

1.8.3.1 Safe Things to Do:

- **Create new files** for your notes and practice work
- **Rename template files** before editing them (like `codebook_yourname.Rmd`)
- **Make new folders** like `my_notes/` or `homework/` for your work
- **Copy lesson files** to a new name before modifying them

1.8.3.2 Things That Cause Git Conflicts:

- **Don't edit the original lesson .Rmd files directly**
- **Don't modify existing file names in the lessons folder**
- **Don't save your work in the main lesson directories**

1.8.3.3 Golden Rule:

Never edit the original lesson files directly. Always create new files for your notes and practice. This way you can always Pull the latest updates without problems!

1.8.3.4 Suggested Workflow:

```
2025_IntroR_and_RStudio/
lessons/                      # DON'T TOUCH - instructor files
img/                          # DON'T TOUCH - course images
_book/                         # DON'T TOUCH - bookdown output
codebook.Rmd                   # DON'T EDIT - copy/rename instead
(other course files)          # DON'T TOUCH - various course materials
my_notes/                      # CREATE - your folder for notes
    lesson1_notes.Rmd
    practice.R
    other_notes.Rmd
codebook_lastname.Rmd  # CREATE - copy of codebook with your name
```

For the codebook: Copy `codebook.Rmd` and rename it to `codebook_lastname.Rmd` (replace “lastname” with your actual last name). This file includes homework questions and space for your notes.

1.9 Step 7: Install Required R Packages

R packages are like apps that add extra features. We need several for this course.

1.9.1 Find the Console:

Look for the **Console** panel in RStudio (bottom left). You'll see a > symbol where you can type.

1.9.2 Install Packages:

First, locate the Console in RStudio: Look at the bottom left panel of RStudio. You'll see a panel labeled "Console" with a > symbol - this is where you type R commands.

Click in the Console and paste this entire block of code (it will take 10-15 minutes):

```
# Install basic tools first
install.packages(c("devtools", "BiocManager", "tidyverse"))

# Install Bioconductor (for biological data analysis)
if (!requireNamespace("BiocManager", quietly = TRUE)) {
  install.packages("BiocManager")
}

# Install all required packages for the course
BiocManager::install(c(
  "apeglm", "bookdown", "clusterProfiler", "DESeq2", "details", "dplyr", "enrichplot", "fansi", "fgsea", "gggraph",
  "ggplot2", "ggrepel", "ggridges", "ggthemes", "gplots", "GO.db", "GOSemSim", "igraph", "knitr", "org.Hs.eg.db",
  "patchwork", "pheatmap", "purrr", "RColorBrewer", "rmarkdown", "rsconnect", "scales", "tidyverse", "tinytex",
  "visNetwork"
))

# Install course-specific package
devtools::install_github("gurinina/GOenrichment", force = TRUE)

# Install document creation tools
if (!tinytex::is_tinytex()) {
  tinytex::install_tinytex(force = TRUE)
}
```

Be patient! This process downloads and installs many packages. You'll see lots of text scrolling by - this is normal.

1.10 Step 8: Verify Everything Works

Let's test that all packages installed correctly:

1.10.1 In the Console, paste and run:

Remember: The Console is the bottom left panel in RStudio with the > symbol.

```
# Check if key packages work
library(ggplot2)
library(dplyr)
library(DESeq2)

# If no error messages appear, you're ready!
cat("Success! All packages are working.\n")
```

1.10.2 Test Creating a Document:

1. In RStudio, go to **File → New File → R Markdown**
2. Leave the default settings and click **OK**
3. Click the “**Knit**” button at the top
4. If a webpage opens with a document, everything is working perfectly!

1.10.3 Creating PDF Documents:

If you want to create PDF files (instead of HTML), you need to modify the document header:

1. **For documents with special characters or emojis:** Change the top of your document to:

```
---
title: "Your Document Title"
output:
  pdf_document:
    latex_engine: xelatex
---
```

2. **For simple documents:** You can use:

```
---
title: "Your Document Title"
output: pdf_document
---
```

Note: If you get errors about Unicode characters when making PDFs, always use `latex_engine: xelatex` in your document header.

1.11 What Success Looks Like

RStudio opens without errors

You can type `2 + 2` in the Console and get `[1] 4`

You can create a new R Markdown document and “Knit” it

You have a folder with course materials from the main repository

1.12 Common Problems and Solutions

1.12.1 “Package not found” errors:

- Make sure you’re typing in the **Console** (bottom left panel)
- Try restarting RStudio and running the code again

1.12.2 “Command not found” in Terminal:

- Close and reopen Terminal
- Try restarting your computer

1.12.3 RStudio can’t find Git:

- Make sure you installed Git before RStudio
- Restart RStudio completely (quit and reopen)

1.12.4 Installation seems stuck:

- Be patient! Package installation can take 15-30 minutes
- As long as you see text appearing, it’s working

1.13 Getting Help

If something doesn’t work:

1. Try restarting RStudio
2. Try restarting your computer
3. Ask a classmate or instructor
4. Email the instructor with:
 - What step you’re on
 - What error message you see (copy and paste it)
 - A screenshot if helpful

Remember: Software installation can be tricky, even for experienced users. Don’t worry if you need help - this is completely normal!

1.14 Next Steps

Once everything is installed:

1. **Explore RStudio** - click around and see what's in each panel
2. **Try the built-in R tutorial**: In the Console, type:

```
install.packages("swirl")
library(swirl)
swirl()
```

3. **Read the course materials** you downloaded
4. **Don't panic!** Learning R takes time, and everyone starts as a beginner

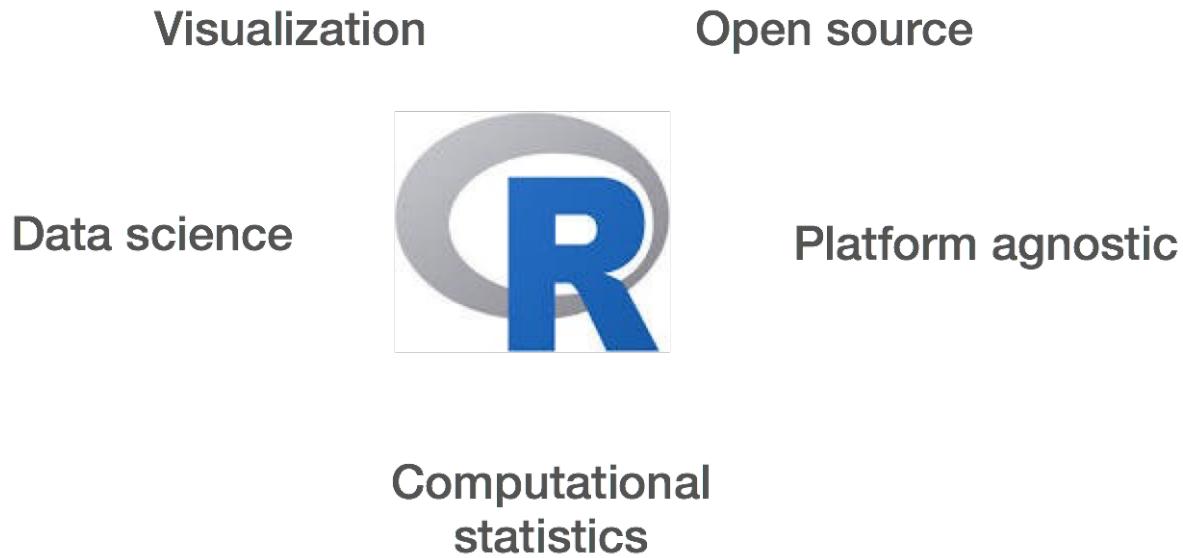
Welcome to R! You're ready to start your data analysis journey.

Chapter 2

Introduction to R and RStudio

2.1 What is R?

R is a programming language and is a great option for analyzing and visualizing data. R is open source and has an active development community that's constantly releasing new packages, making R code even easier to use.



2.2 What is RStudio?

RStudio is an integrated development environment (IDE) for R.



2.3 Open a project directory in RStudio

1. Start Rstudio, and go to the `File` menu, select `Open Project`, and navigate to your `2025_IntroR_and_RStudio` folder and double click on `2024_IntroR_and_RStudio.Rproj`.
2. When RStudio opens, you will see four panels in the window.
3. Open the file `17-introR_codebook.Rmd` in the lessons directory under the `File` menu (lower right panel). It will open in the script window in the upper left. This file contains all the R code that we will be running in the lessons.

2.4 RStudio Interface

The RStudio interface has four main panels:

1. **Console:** (lower left panel) where you can type commands and see output.
2. **Script editor:** (upper left panel) where you can type out commands and save to file. You can also submit the commands to run in the console.
3. **Environment/History/Git:** (upper right panel)
 - Environment: lists the active objects in your R session
 - History: keeps track of all commands run in console.
 - Git keeps track of any changes in your git repository. It is important that you don't change any of the original files in your working directory. In fact, it's best if you save `17-introR_codebook.Rmd` which contains all the code we will be running under another name, e.g. `17-codebook_yourname.Rmd`, with your name at the end. You can save this either in the lessons directory or in the scripts directory, this is the document you will be handing in as your homework for this module. This way you can always go back to the original file if you need to. If you want to take notes, you can do so in the `17-codebook_yourname.Rmd` file.
4. **Files/Plots/Packages/Help** (lower right panel)
 - File: lists all the files in your project directory (current directory)
 - Plots: shows the output of graphs you generate – set the output of your Rmd file to “Preview in Viewer Pane”; cogwheel next to Knitr button at the top of your window.
 - Packages: lists the loaded libraries
 - Help: interface for R help menu for functions and packages

2.5 Interacting with R

There are **two main ways** of interacting with R in RStudio: using the **console** or by using **script editor** (plain text files that contain your code).

2.5.1 Console window

The **console window** (in RStudio, the bottom left panel) is the place where R will show the results of a command. You can type commands directly into the console, but they will be forgotten when you close the session.

Let's test it out; in your console type:

```
3 + 5
```

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

```
> 3+5
[1] 8
> |
```

2.5.2 Script editor

You can use the comments character `#` to add additional descriptions. It's best practice to comment liberally to describe the commands you are running using `#`. To run the code, you click on the little green arrow on the side of the `code` chunk. Let's run the second chunk.

```
# Intro to R Lesson
## I am adding 3 and 5. R is fun!
3+5
```

You will see the command run in the console and output the result.

You can also run the code by highlighting it and pressing the **Ctrl** and **Return/Enter** keys at the same time as a shortcut.

```
Console - /Desktop/Intro to R/ 
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 3+5
[1] 8
> # Intro to R Lesson
> # Feb 16th, 2016
>
> # Interacting with R
>
> ## I am adding 3 and 5. R is fun!
> 3+5
[1] 8
>
```

2.6 The R syntax

Now that we know how to talk with R via the script editor or the console, we want to use R for something more than adding numbers. To do this, we need to know more about the R syntax.

The main parts of R (syntax) include:

- the **comments #** and how they are used to document function and its content
- **variables** and **functions**
- the **assignment operator <-**
- the **=** for **arguments** in functions

We will go through each of these in more detail, starting with the assignment operator.

2.7 Assignment operator

To do useful and interesting things in R, we need to assign *values* to *variables* using the assignment operator, **<-**. For example, we can use the assignment operator to assign the value of 3 to **x** by executing:

```
x <- 3
```

The assignment operator (**<-**) assigns **values on the right to variables on the left**.

In Windows, typing Alt + - (push Alt at the same time as the - key, on Mac type option + -) will write <- in a single keystroke.

2.8 Variables

In the example above, we created a variable called `x` and assigned it a value of 3.

Let's create another variable called `y` and give it a value of 5.

```
y <- 5
```

You can view information on the variable by looking in your `Environment` window in the upper right-hand corner of the RStudio interface.

The screenshot shows the RStudio interface with the 'Environment' tab selected. The window title is 'Values'. It contains two entries: 'x' with the value '3' and 'y' with the value '5'. There are also buttons for 'Import Dataset', 'Clear', and 'Global Environment'.

x	3
y	5

Now we can reference these variables by name to perform mathematical operations on the values contained within. What do you get in the console for the following operation:

```
x + y
```

Try assigning the results of this operation to another variable called `number`.

```
number <- x + y
```

Exercises

1. Try changing the value of the variable `x` to 5. What happens to `number`?
 2. Now try changing the value of variable `y` to contain the value 10. What do you need to do, to update the variable `number`?
-

2.8.1 Tips on variable names

Variables can be given almost any name, such as `x`, `current_temperature`, or `subject_id`. However, there are some rules / suggestions you should keep in mind:

- Avoid names starting with a number (`2x` is not valid but `x2` is)
- Avoid dots (.) within a variable name; dots have a special meaning in R (for methods) so it's best to avoid them.

- Keep in mind that **R is case sensitive** (e.g., `genome_length` is different from `Genome_length`)
- Be consistent with the styling of your code (where you put spaces, how you name variable, etc.). In R, two popular style guides:
 - Hadley Wickham's style guide
 - Google's.

2.9 Interacting with data in R

R is commonly used for handling big data, and so it only makes sense that we learn about R in the context of some kind of relevant data.

2.9.1 Data directory

You can access the files we need for this workshop in your data directory. **We will discuss these files a bit later in the lesson.**

2.9.2 The mouse dataset

In this example dataset we have collected whole brain samples from 12 mice and want to evaluate expression differences between them. The expression data represents normalized count data (`normalized_counts.txt`) obtained from RNA-sequencing of the 12 brain samples. This data is stored in a comma separated values (CSV) file as a 2-dimensional matrix, with **each row corresponding to a gene and each column corresponding to a sample**.

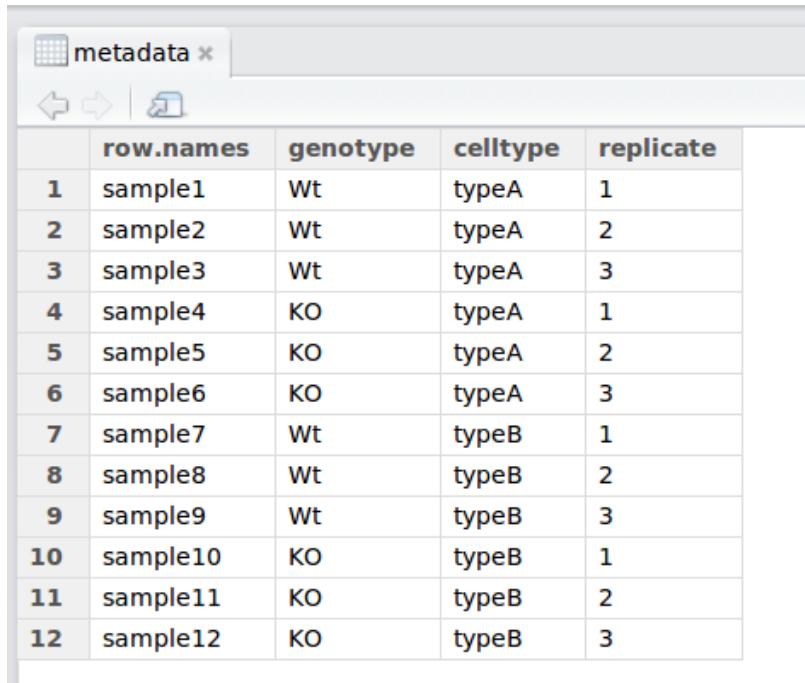
counts x													38,828 observations of 12 variables	
	row.names	sample2	sample5	sample7	sample8	sample9	sample4	sample6	sample12	sample3	sample11	sample10	sample1	
1	ENSMUSG000000000001	1.92650e+01	2.37222e+01	2.61161e+00	5.849540e+00	6.51263e+00	2.40767e+01	2.08198e+01	2.69158e+01	2.08895e+01	2.40465e+01	2.41981e+01	1.97848e+01	
2	ENSMUSG000000000003	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	
3	ENSMUSG000000000028	1.03229e+01	8.26954e-01	1.13441e+00	6.987504e-01	9.25117e-01	8.27891e-01	1.16863e+00	6.73563e-01	8.92183e-01	9.75327e-01	1.04592e+00	9.37792e-01	
4	ENSMUSG000000000031	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	5.97726e-02	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	
5	ENSMUSG000000000037	5.60330e-02	4.73238e-02	0.00000e+00	6.859390e-02	4.94147e-02	1.80883e-01	1.43884e-01	6.62324e-02	1.46196e-01	2.06405e-02	1.70040e-02	1.51417e-01	
6	ENSMUSG000000000043	2.58134e-01	1.07302e+00	2.52342e-01	2.970320e-01	2.08280e-01	2.19172e+00	1.68538e+00	1.16197e-01	4.21286e-01	6.34322e-02	3.69550e-01	2.56733e-01	
7	ENSMUSG000000000056	6.04799e+00	6.41163e+00	7.39490e+00	8.024460e+00	8.16998e+00	5.62012e+00	5.51777e+00	7.86054e+00	6.02025e+00	8.60562e+00	8.49791e+00	5.99981e+00	
8	ENSMUSG000000000058	3.97181e+00	5.21366e+00	1.13985e+00	6.411750e-01	6.29602e-01	7.04591e+00	7.39511e+00	1.38085e+00	6.16145e+00	1.01689e+00	1.58082e+00	5.27847e+00	
9	ENSMUSG000000000078	3.23030e+01	3.56341e+00	9.94056e+00	7.533320e+00	1.07182e+01	4.26057e+00	5.38282e+00	1.78086e+01	3.12496e+01	1.41614e+01	1.79056e+01	3.72141e+01	
10	ENSMUSG000000000085	2.29507e+01	1.79712e+01	1.109130e+01	1.24428e+01	1.68254e+01	1.77215e+01	2.40850e+01	2.14156e+01	1.92192e+01	1.96968e+01	2.54044e+01		
11	ENSMUSG000000000088	1.13552e+02	1.95870e+02	2.52809e+02	8.539790e+02	8.30703e+02	2.23153e+02	1.93232e+02	1.39378e+02	9.93445e+01	1.35738e+02	1.28969e+02	1.14664e+02	
12	ENSMUSG000000000093	1.47470e+00	2.01829e-01	1.76900e+00	2.419240e+00	4.18875e+00	1.64318e-01	1.97125e-01	9.87224e-01	1.06196e+00	4.95501e-01	6.24965e-01	7.22611e-01	
13	ENSMUSG000000000094	5.29831e-02	2.23309e-01	0.00000e+00	5.563000e-03	1.47304e-02	4.13359e-02	2.78842e-01	9.19884e-03	7.34683e-02	0.00000e+00	2.55498e-02	5.14853e-03	
14	ENSMUSG000000000103	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	
15	ENSMUSG000000000120	3.09596e+02	6.65014e+02	5.22346e+01	7.018820e+01	6.34166e+01	8.73788e+00	9.63375e+00	1.60367e+02	2.97233e+02	1.55449e+02	1.77215e+02	3.16047e+02	
16	ENSMUSG000000000125	4.31764e-01	1.44202e-02	0.00000e+00	4.286370e-02	3.73345e-02	1.67966e-02	0.00000e+00	3.30389e-01	2.98720e-01	3.90483e-01	1.36312e-01	3.64900e-01	
17	ENSMUSG000000000126	3.55276e+00	5.27728e+00	6.37010e-01	1.088940e+00	9.15849e-01	5.45301e+00	4.16439e+00	5.21701e+00	3.93439e+00	6.35234e+00	5.76519e+00	4.64958e+00	
18	ENSMUSG000000000127	7.22679e+00	8.31334e+00	3.04145e+00	2.572820e+00	3.04853e+00	1.15169e+01	8.69086e+00	1.01329e+01	7.56826e+00	9.26286e+00	9.69557e+00	8.28828e+00	
19	ENSMUSG000000000131	6.71003e+01	5.07176e+01	4.892R0e+01	7.260810e+01	3.99663e+01	4.6627Re+01	5.14121e+01	7.87299e+01	6.62407e+01	7.33902e+01	6.81717e+01	5.98459e+01	

2.9.3 The metadata

We have another file in which we identify **information about the data or metadata** (`mouse_exp_design.csv`). Our metadata is also stored in a CSV file. In this file, each row corresponds to a sample and each column contains some information about each sample.

The first column contains the row names, and **note that these are identical to the column names in our expression data file above** (albeit, in a slightly different order). The next few columns contain information about our samples that allow us to categorize them. For example, the second column contains genotype information for each sample. Each sample

is classified in one of two categories: Wt (wild type) or KO (knockout). *What types of categories do you observe in the remaining columns?*



The screenshot shows a Jupyter Notebook cell with the title "metadata". The cell contains a table with four columns: "row.names", "genotype", "celltype", and "replicate". The data consists of 12 rows, indexed from 1 to 12, representing samples. The "genotype" column has values "Wt" and "KO". The "celltype" column has values "typeA" and "typeB". The "replicate" column has values 1, 2, and 3, repeated three times each.

	row.names	genotype	celltype	replicate
1	sample1	Wt	typeA	1
2	sample2	Wt	typeA	2
3	sample3	Wt	typeA	3
4	sample4	KO	typeA	1
5	sample5	KO	typeA	2
6	sample6	KO	typeA	3
7	sample7	Wt	typeB	1
8	sample8	Wt	typeB	2
9	sample9	Wt	typeB	3
10	sample10	KO	typeB	1
11	sample11	KO	typeB	2
12	sample12	KO	typeB	3

These lessons have been developed by members of the teaching team at the Harvard Chan Bioinformatics Core (HBC). These are open access materials distributed under the terms of the Creative Commons Attribution license (CC BY 4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Chapter 3

R syntax and data structures

3.1 Data Types

Variables can contain values of specific types within R. The **data types** that R uses include:

- "numeric" for any numerical value, including whole numbers and decimals.
- "character" for text values, denoted by using quotes (" ") around value.
- "integer" for whole numbers (e.g., 2L, the L indicates to R that it's an integer). It behaves similar to the **numeric** data type for most tasks or functions.
- "logical" datatypes are **TRUE** and **FALSE** in all capital letters (the Boolean data type). The **logical** data type can also be specified using T for **TRUE** in all capital letters, and F for **FALSE**. T and F are not recommended for use in R, as they can be confused with other functions or variables.

The table below provides examples of each of the commonly used data types:

Data Type	Examples
Numeric:	1, 1.5, 20, pi
Character:	"anytext", "5", "TRUE"
Integer:	2L, 500L, -17L
Logical:	TRUE, FALSE, T, F

3.2 Data Structures

So far we have seen variables with a single value. **Variables can store more than just a single value, they can store a multitude of different data structures.** These include, but are not limited to, vectors (**c**), factors (**factor**), matrices (**matrix**) and data frames (**data.frame**).

3.2.1 Vectors

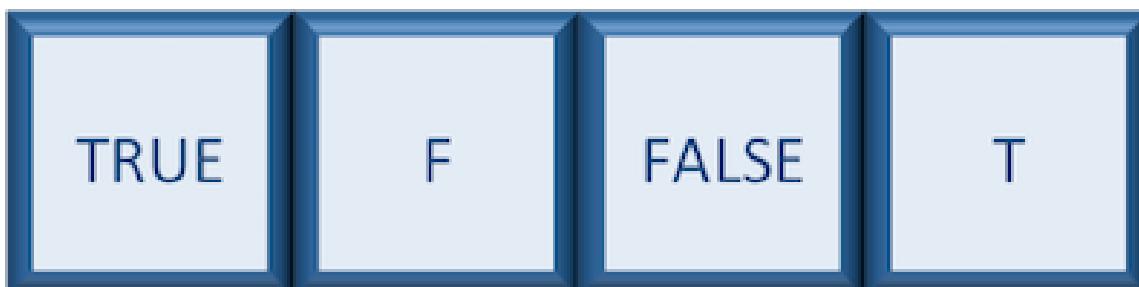
A vector is the most common and basic data structure in R, and is pretty much the workhorse of R. It's basically just a collection of values, mainly either numbers:



or characters:

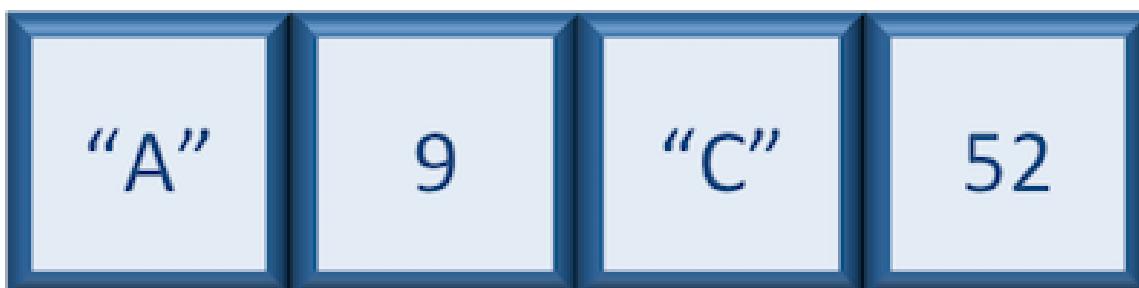


or logical values:

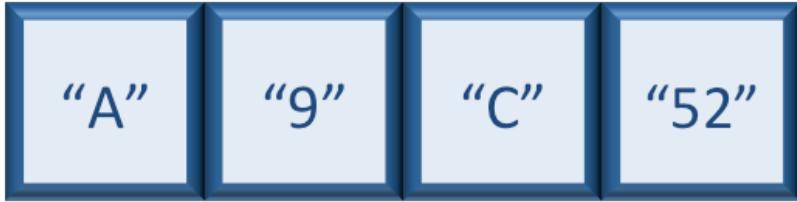


Note that all values in a vector must be of the same data type. If you try to create a vector with more than a single data type, R will try to coerce it into a single data type.

For example, if you were to try to create the following vector:



R will coerce it into:



The values in a vector are called *elements*.

Each **element** contains a single value, and there is no limit to how many elements you can have. A vector is assigned to a single variable, because regardless of how many elements it contains, in the end it is still a vector.

Let's create a vector of genome lengths and assign it to a variable called `glengths`.

Each element of this vector contains a single numeric value, and three values will be combined together into a vector using `c()` (the combine function). All of the values are put within the parentheses and separated with a comma.

```
# Create a numeric vector and store the vector as a variable called 'glengths'
glengths <- c(4.6, 3000, 50000)
glengths
```

Note your environment shows the `glengths` variable is numeric (num) and tells you the `glengths` vector starts at element 1 and ends at element 3 (i.e. your vector contains 3 values) as denoted by the [1:3].

A vector can also contain characters. Create another vector called `species` with three elements, where each element corresponds with the genome sizes vector (in Mb).

```
# Create a character vector and store the vector as a variable called 'species'
species <- c("ecoli", "human", "corn")
species
```

Exercise

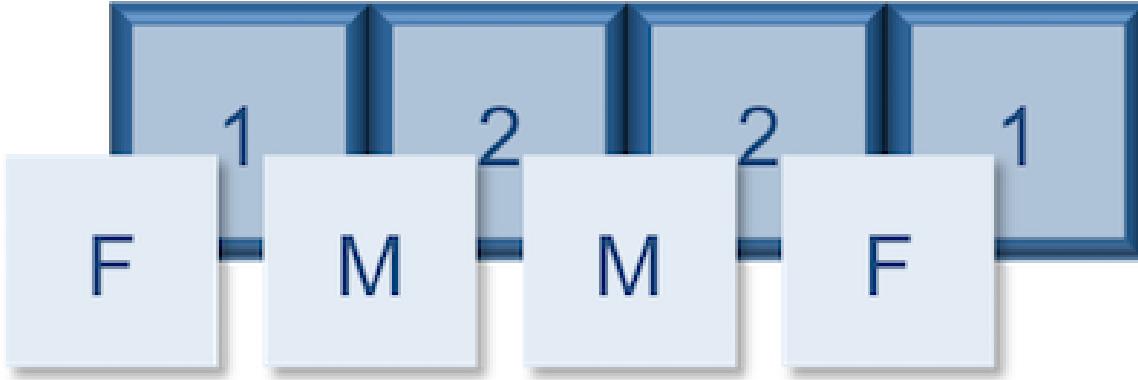
Try to create a vector of numeric and character values by *combining* the two vectors that we just created (`glengths` and `species`). Assign this combined vector to a new variable called `combined`. Hint: you will need to use the combine `c()` function to do this. Print the `combined` vector in the console, what looks different compared to the original vectors? What do you notice about the output of the `combined` vector?

3.2.2 Factors

A **factor** is a special type of vector that is used to **store categorical data**. Each unique category is referred to as a **factor level** (i.e. category = level).

For instance, if we have four animals and the first animal is female, the second and third are male, and the fourth is female, we could create a factor that appears like a vector, but has integer values stored under-the-hood. The integer value assigned is a one for females and a two for males. The numbers are assigned in alphabetical order, so because the f- in females comes

before the m- in males in the alphabet, females get assigned a one and males a two. In later lessons we will show you how you could change these assignments.



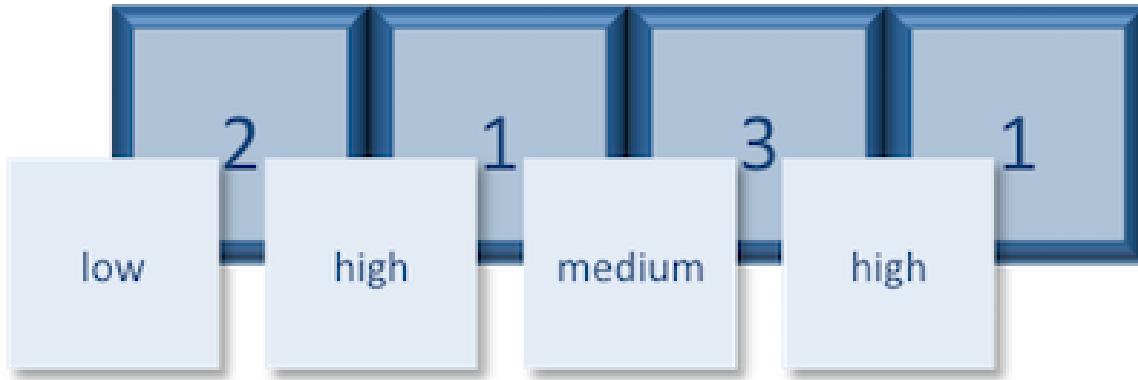
Let's create a factor vector and explore a bit more. We'll start by creating a character vector describing three different levels of expression. Perhaps the first value represents expression in mouse1, the second value represents expression in mouse2, and so on and so forth:

```
# Create a character vector and store the vector as a variable called 'expression'
expression <- c("low", "high", "medium", "high", "low", "medium", "high")
```

Now we can convert this character vector into a *factor* using the **factor()** function:

```
# Turn 'expression' vector into a factor
expression <- factor(expression)
```

So, what exactly happened when we applied the **factor()** function?



The expression vector is categorical, in that all the values in the vector belong to a set of categories; in this case, the categories are **low**, **medium**, and **high**.

So now that we have an idea of what factors are, when would you ever want to use them?

Factors are extremely valuable for many operations often performed in R and are necessary for many statistical methods, as you'll see. As an example, if you want to color your plots by treatment type, then you would need the treatment variable to be a factor.

Exercises

Let's say that in our experimental analyses, we are working with three different sets of cells: normal, cells knocked out for geneA (a very exciting gene), and cells overexpressing geneA. We have three replicates for each celltype.

1. Create a vector named `samplegroup` with nine elements: 3 control (“CTL”) values, 3 knock-out (“KO”) values, and 3 over-expressing (“OE”) values.
 2. Turn `samplegroup` into a factor data structure.
-

3.2.3 Matrix

A `matrix` in R is a collection of vectors of **same length and identical datatype**. Vectors can be combined as columns in the matrix or by row, to create a 2-dimensional structure and are usually of numeric datatype.

90	5	137	9
87	40	2	52
4	102	32	41

3.2.4 Data Frame

A `data.frame` is the *de facto* data structure for most tabular data and what we use for statistics and plotting. A `data.frame` is similar to a matrix in that it's a collection of vectors of the **same length** and each vector represents a column. However, in a data frame **each vector can be of a different data type** (e.g., characters, integers, factors). In the data frame pictured below, the first column is character, the second column is numeric, the third is character, and the fourth is logical.

“A”	102	“Hela”	TRUE
“B”	40	“BHK”	F
“C”	12	“hESC”	T

A data frame is the most common way of storing data in R, and if used systematically makes data analysis easier.

We can create a dataframe by bringing **vectors** together to **form the columns**. We do this using the `data.frame()` function, and giving the function the different vectors we would like to bind together. *This function will only work for vectors of the same length.*

```
# Create a data frame and store it as a variable called 'df'
df <- data.frame(species, glengths)
```

We can see that a new variable called `df` has been created in our **Environment** within a new section called **Data**. In the **Environment**, it specifies that `df` has 3 observations of 2 variables. What does that mean? In R, rows always come first, so it means that `df` has 3 rows and 2 columns.

Exercise

Create a data frame called `favorite_books` with the following vectors as columns:

```
titles <- c("Catch-22", "Pride_and_Prejudice", "Nineteen_Eighty_Four")
pages <- c(453, 432, 328)
```

Chapter 4

Functions in R

4.1 Functions and their arguments

4.1.1 What are functions?

A key feature of R is functions. Functions are “self contained” modules of code that accomplish a specific task. Functions usually take in some sort of data structure (value, vector, dataframe etc.) as arguments, process them, and then return a result.

The general usage for a function is the name of the function followed by parentheses:

```
function_name(input)
```

The input(s) are called **arguments**, which can include:

1. the physical object (any data structure) on which the function carries out a task
2. specifications that alter the way the function operates (e.g. options)

Most functions can take several arguments. If you don’t specify a required argument when calling the function, you will receive an error unless the function has set a default value for the argument.

4.1.2 Basic functions

We have already used a few examples of basic functions in the previous lessons i.e `c()`, and `factor()`. These functions are available as part of R’s built in capabilities, and we will explore a few more of these base functions below.

Many of the base functions in R involve mathematical operations. One example would be the function `sqrt()`. The input/argument must be a number, and the output is the square root of that number. Let’s try finding the square root of 81:

```
sqrt(81)
```

Now what would happen if we **called the function** (e.g. ran the function), on a *vector of values* instead of a single value?

```
sqrt(glengths)
```

In this case the task was performed on each individual value of the vector `glengths` and the respective results were displayed.

Let's try another function, this time using one that we can change some of the *options* (arguments that change the behavior of the function), for example `round`:

```
round(3.14159)
```

We can see that we get `3`. That's because the default is to round to the nearest whole number. **What if we want a different number of significant digits?** Let's first learn how to find available arguments for a function.

4.1.3 Seeking help on arguments for functions

The best way of finding out this information is to use the `?` followed by the name of the function. Doing this will open up the help manual in the bottom right panel of RStudio that will provide a description of the function, usage, arguments, details, and examples:

```
?round
```

Alternatively, if you are familiar with the function but just need to remind yourself of the names of the arguments, you can use:

```
args(round)
```

Even more useful is the `example()` function. This will allow you to run the examples section from the Online Help to see exactly how it works when executing the commands. Let's try that for `round()`:

```
example("round")
```

In our example, we can change the number of digits returned by **adding an argument**. We can type `digits=2` or however many we may want:

```
round(3.14159, digits=2)
```

Exercise

1. Let's use base R function to calculate **mean** value of the `glengths` vector. You might need to search online to find what function can perform this task.
2. Create a new vector `test <- c(1, NA, 2, 3, NA, 4)`. Use the same base R function from exercise 1 (with addition of proper argument), and calculate mean value of the `test` vector. The output should be `2.5`.

NOTE: In R, missing values are represented by the symbol `NA` (not available). It's a way to make sure that users know they have missing data, and make a conscious decision on how to deal with it. There are ways to ignore NA during statistical calculation, or to remove NA from the vector.

3. Another commonly used base function is `sort()`. Use this function to sort the `glengths` vector in **descending** order.

4.1.4 User-defined Functions

One of the great strengths of R is the user's ability to add functions. Sometimes there is a small task (or series of tasks) you need done and you find yourself having to repeat it multiple times. In these types of situations, it can be helpful to create your own custom function. The **structure of a function is given below:**

```
name_of_function <- function(argument1, argument2) {
  statements or code that does something
  return(something)
}
```

- First you give your function a name.
- Then you assign value to it, where the value is the function.

When **defining the function** you will want to provide the **list of arguments required** (inputs and/or options to modify behaviour of the function), and wrapped between curly brackets place the **tasks that are being executed on/using those arguments**. The argument(s) can be any type of object (like a scalar, a matrix, a dataframe, a vector, a logical, etc), and it's not necessary to define what it is in any way.

Finally, you can “**return**” the **value of the object from the function**, meaning pass the value of it into the global environment. The important idea behind functions is that objects that are created within the function are local to the environment of the function – they don't exist outside of the function.

Let's try creating a simple example function. This function will take in a numeric value as input, and return the squared value.

```
square_it <- function(x) {
  square <- x * x
  return(square)
}
```

Once you run the code, you should see a function named **square_it** in the Environment panel (located at the top right of Rstudio interface). Now, we can use this function as any other base R functions. We type out the name of the function, and inside the parentheses we provide a numeric value **x**:

```
square_it(5)
```

Pretty simple, right? In this case, we only had one line of code that was run, but in theory you could have many lines of code to get obtain the final results that you want to “return” to the user.

We have only scratched the surface here when it comes to creating functions! If you are interested you can also find more detailed information on writing functions R-bloggers site.

Exercise

1. Write a function called **multiply_it**, which takes two inputs: a numeric value **x**, and a numeric value **y**. The function will return the product of these two numeric values, which is **x * y**. For example, **multiply_it(x=4, y=6)** will return output 24.

Chapter 5

Packages and libraries

5.1 Packages and Libraries

Packages are collections of R functions, data, and compiled code in a well-defined format, created to add specific functionality.

There are a set of **standard (or base) packages** which are considered part of the R source code and automatically available as part of your R installation. Base packages contain the **basic functions** that allow R to work, and enable standard statistical and graphical functions on datasets; for example, all of the functions that we have been using so far in our examples.

The terms *package* and *library* are sometimes used synonymously.

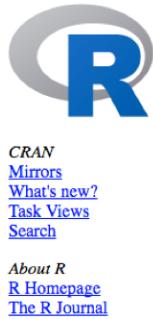
You can check what libraries are loaded in your current R session by typing into the console:

```
sessionInfo() #Print version information about R, the OS and attached or loaded packages  
# OR  
search() #Gives a list of attached packages
```

As you work with R, you'll see that there are thousands of R packages that offer a wide variety of functionality. Many packages can be installed from the CRAN or Bioconductor repositories.

5.1.1 Package installation from CRAN

CRAN is a repository where the latest downloads of R are found in addition to source code for different user contributed R packages.



Available CRAN Packages By Name

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

A3	Accurate, Adaptable, and Accessible Error Metrics for Predictive Models
abbyyR	Access to Abbyy Optical Character Recognition (OCR) API
abc	Tools for Approximate Bayesian Computation (ABC)
ABCanalysis	Computed ABC Analysis
abc.data	Data Only: Tools for Approximate Bayesian Computation (ABC)
abcdeFBA	ABCDE_FBA: A-Biologist-Can-Do-Everything of Flux Balance Analysis with this package
ABCOptim	Implementation of Artificial Bee Colony (ABC) Optimization
ABCp2	Approximate Bayesian Computational Model for Estimating P2
abcrf	Approximate Bayesian Computation via Random Forests

Packages for R can be installed from the CRAN package repository using the `install.packages` function.

An example is given below for the `ggplot2` package that will be required for some plots we will create later on. Run this code to install `ggplot2`.

```
install.packages("ggplot2")
```

5.1.2 Package installation from Bioconductor

Alternatively, packages can also be installed from Bioconductor, another repository of packages which provides tools for the analysis and comprehension of high-throughput **genomic data**.



You should already have installed Bioconductor, by installing BiocManager. *This only needs to be done once ever for your R installation.*

```
# DO NOT RUN THIS!
# install.packages("BiocManager")
```

Now you can use the `install()` function from the `BiocManager` package to install a package by providing the name in quotations.

Here we have the code to install `ggplot2`, through Bioconductor:

```
# DO NOT RUN THIS!
BiocManager::install("ggplot2")
```

The code above may not be familiar to you - it is essentially using a new operator, a double colon `::` to execute a function from a particular package. This is the syntax: `package::function_name()`. It's used to avoid conflicts in functions from different packages.

5.1.3 Loading libraries

Once you have the package installed, you can **load the library** into your R session for use. Any of the functions that are specific to that package will be available for you to use by simply calling the function as you would for any of the base functions. *Note that quotations are not required here.*

```
library(ggplot2)
```

We only need to install a package once on our computer. However, to use the package, we need to load the library every time we start a new R/RStudio environment.



Images sourced from <https://www.wikihow.com/Change-a-Light-Bulb>

5.1.4 Finding functions specific to a package

This is your first time using `ggplot2`, how do you know where to start and what functions are available to you? One way to do this, is by using the **Package** tab in RStudio. If you click on the tab, you will see listed all packages that you have installed. For those *libraries that you have loaded*, you will see a blue checkmark in the box next to it. Scroll down to `ggplot2` in your list:

Name	Description	Ver...
<input checked="" type="checkbox"/> ggplot2	An Implementation of the Grammar of Graphics	2.1.0
<input type="checkbox"/> git2r	Provides Access to Git Repositories	0.15.0
<input type="checkbox"/> GO.db	A set of annotation maps describing the entire Gene Ontology	3.1.2
<input type="checkbox"/> GOSemSim	GO-terms Semantic Similarity	1.26.0

If your library is successfully loaded you will see the box checked, as in the screenshot above. Now, if you click on `ggplot2` RStudio will open up the help pages and you can scroll through.

Other resources

An alternative is to find the help manual online, which can be less technical and sometimes easier to follow. For example, this website is much more comprehensive for `ggplot2` and is the result of a Google search.

If you can't find what you are looking for, you can use the rdrr.io website that searches through the help files across all packages available. It also provides source code for functions.

To see the functions in a package you can also type:

```
ls("package:ggplot2")
```

Exercise

The `ggplot2` package is part of the `tidyverse` suite of integrated packages which was designed to work together to make common data science operations more user-friendly. **We will be using the `tidyverse` suite in later lessons, and so let's install it using `BiocManager`.**

Chapter 6

Subsetting: vectors and factors

6.1 Selecting data using indices

When analyzing data, we often want to **partition the data so that we are only working with selected columns or rows**. A data frame or data matrix is simply a collection of vectors combined together. So let's begin with vectors and how to access different elements, and then extend those concepts to dataframes.

6.1.1 Vectors

6.1.1.1 Selecting using indices

If we want to extract one or several values from a vector, we must provide one or several indices using square brackets []. The **index represents the element number within a vector**. R indices start at 1.

Let's start by creating a vector called age:

```
age <- c(15, 22, 45, 52, 73, 81)
```



Suppose we only wanted the fifth value of this vector, we would use the following syntax:

```
age[5]
```

If we wanted all values except the fifth value of this vector, we would use the following:

```
age[-5]
```

If we wanted to select more than one element we would still use the square bracket syntax, but rather than using a single value we would pass in a *vector of several index values*:

```
age[c(3,5,6)]    ## nested

# OR

## create a vector first then select
idx <- c(3,5,6) # create vector of the elements of interest
age[idx]
```

To select a sequence of continuous values from a vector, we would use `:` which is a special function that creates numeric vectors of integer in increasing or decreasing order. Let's select the *first four values* from `age`:

```
age[1:4]
```

Alternatively, if you wanted the reverse could try `4:1` for instance, and see what is returned.

Exercises

1. Create a vector called `alphabets` with the following letters, C, D, X, L, F.
 2. Use the associated indices along with `[]` to do the following:
 - only display C, D and F
 - display all except X
 - display the letters in the opposite order (F, L, X, D, C)
-

6.1.1.2 Selecting using indices with logical operators

We can also use indices with logical operators. Logical operators include greater than (`>`), less than (`<`), and equal to (`==`). A full list of logical operators in R is displayed below:

Operator	Description
<code>></code>	greater than
<code>>=</code>	greater than or equal to
<code><</code>	less than
<code><=</code>	less than or equal to
<code>==</code>	equal to
<code>!=</code>	not equal to
<code>&</code>	and
<code> </code>	or

We can use logical expressions to determine whether a particular condition is true or false. For example, let's use our `age` vector:

```
age
```

If we wanted to know if each element in our age vector is greater than 50, we could write the following expression:

```
age > 50
```

Returned is a vector of logical values the same length as age with TRUE and FALSE values indicating whether each element in the vector is greater than 50.

```
[1] FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

We can use these logical vectors to select only the elements in a vector with TRUE values at the same position or index as in the logical vector.

Select all values in the age vector over 50 **or** age less than 18:

```
age > 50 | age < 18
age
age[age > 50 | age < 18]  ## nested
# OR
## create a vector first then select
idx <- age > 50 | age < 18
age[idx]
```

6.1.1.3 Indexing with logical operators using the which() function

While logical expressions will return a vector of TRUE and FALSE values of the same length, we could use the **which()** function to output the indices where the values are TRUE. For example:

```
which(age > 50 | age < 18)
age[which(age > 50 | age < 18)]  ## nested
# OR
## create a vector first then select
idx_num <- which(age > 50 | age < 18)
age[idx_num]
```

Notice that we get the same results regardless of whether or not we use the **which()**.

6.1.2 Factors

Since factors are special vectors, the same rules for selecting values using indices apply. The elements of the expression factor created previously had the following categories or levels: low, medium, and high. Let's extract the values of the factor with high expression, and let's use nesting here:

```
expression[expression == "high"]
## This will only return those elements in the factor equal to "high"
```

Exercise

Extract only those elements in `samplegroup` that are not KO (*nesting the logical operation is optional*).

6.1.2.1 Releveling factors

We have briefly talked about factors, but this data type only becomes more intuitive once you've had a chance to work with it. Let's take a slight detour and learn about how to **relevel categories within a factor**.

To view the integer assignments under the hood you can use `str()`:

```
expression

str(expression)
Factor w/ 3 levels "high", "low", "medium": 2 1 3 1 2 3 1
```

The categories are referred to as “factor levels”. As we learned earlier, the levels in the `expression` factor were assigned integers alphabetically, with high=1, low=2, medium=3. However, it makes more sense for us if low=1, medium=2 and high=3, i.e. it makes sense for us to “relevel” the categories in this factor.

To relevel the categories, you can add the `levels` argument to the `factor()` function, and give it a vector with the categories listed in the required order:

```
expression <- factor(expression, levels=c("low", "medium", "high"))
# you can re-factor a factor

str(expression)
Factor w/ 3 levels "low", "medium", ... : 1 3 2 3 1 2 3
```

Now we have a leveled factor with low as the lowest or first category, medium as the second and high as the third. This is reflected in the way they are listed in the output of `str()`, as well as in the numbering of which category is where in the factor.

Note: Releveling becomes necessary when you need a specific category in a factor to be the “base” category, i.e. category that is equal to 1. One example would be if you need the “control” to be the “base” in a given RNA-seq experiment.

Exercise

Use the `samplegroup` factor we created in a previous lesson, and relevel it such that KO is the first level followed by CTL and OE.

Chapter 7

Reading and data inspection

7.1 Reading data into R

7.1.1 The basics

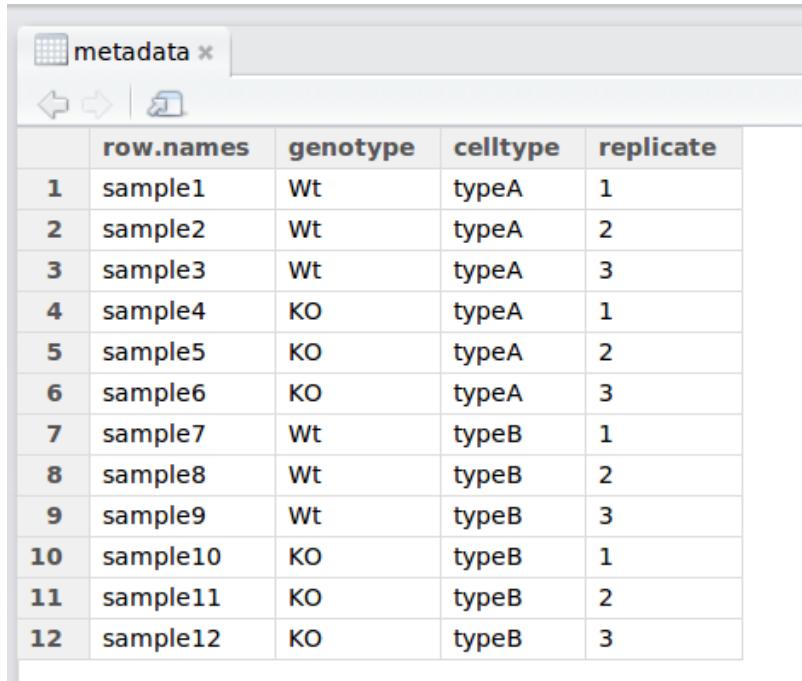
There are several function in base R that exist to read data in, and the function in R you use will depend on the file format being read in. Below we have a table with the base R functions that can be used for importing some common text data types (plain text).

Data type	Extension	Function
Comma separated values	csv	<code>read.csv()</code>
Tab separated values	tsv	<code>read.delim</code>
Other delimited formats	txt	<code>read.table()</code>

For example, if we have text file where the columns are separated by commas (comma-delimited), you could use the function `read.csv`. However, if the data are separated by a different delimiter in a text file (e.g. `:`, `;`, ` `), you could use the generic `read.table` function and specify the delimiter (`sep = " "`) as an argument in the function.

7.1.2 Metadata

When working with large datasets, you will very likely be working with a “metadata” file which contains the information about each sample in your dataset.



	row.names	genotype	celltype	replicate
1	sample1	Wt	typeA	1
2	sample2	Wt	typeA	2
3	sample3	Wt	typeA	3
4	sample4	KO	typeA	1
5	sample5	KO	typeA	2
6	sample6	KO	typeA	3
7	sample7	Wt	typeB	1
8	sample8	Wt	typeB	2
9	sample9	Wt	typeB	3
10	sample10	KO	typeB	1
11	sample11	KO	typeB	2
12	sample12	KO	typeB	3

7.1.3 The `read.csv()` function

Let's bring in the metadata file in our data folder (`mouse_exp_design.csv`) using the `read.csv` function.

First, check the arguments for the function using the `?read.csv` command:

```
?read.csv
```

Description

Reads a file in table format and creates a data frame from it, with cases corresponding to lines and variables to fields in the file.

Usage

```
read.table(file, header = FALSE, sep = "", quote = "\"\"",  
          dec = ".", numerals = c("allow.loss", "warn.loss", "no.loss"),  
          row.names, col.names, as.is = !stringsAsFactors,  
          na.strings = "NA", colClasses = NA, nrow = -1,  
          skip = 0, check.names = TRUE, fill = !blank.lines.skip,  
          strip.white = FALSE, blank.lines.skip = TRUE,  
          comment.char = "#",  
          allowEscapes = FALSE, flush = FALSE,  
          stringsAsFactors = default.stringsAsFactors(),  
          fileEncoding = "", encoding = "unknown", text, skipNul = FALSE)  
  
read.csv(file, header = TRUE, sep = ",", quote = "\"\"",  
        dec = ".", fill = TRUE, comment.char = "", ...)  
  
read.csv2(file, header = TRUE, sep = ";", quote = "\"\"",  
         dec = ",", fill = TRUE, comment.char = "", ...)  
  
read.delim(file, header = TRUE, sep = "\t", quote = "\"\"",  
          dec = ".", fill = TRUE, comment.char = "", ...)  
  
read.delim2(file, header = TRUE, sep = "\t", quote = "\"\"",  
            dec = ",", fill = TRUE, comment.char = "", ...)
```

The first item on the documentation page is the function **Description**, which specifies that the output of this set of functions is going to be a **data frame**.

In usage, all of the arguments listed for `read.table()` are the default values for all of the family members unless otherwise specified for a given function. Let's take a look at 3 examples:

1. The separator

- for `read.table() sep = ""` (space or tab)
- for `read.csv() sep = ","` (a comma).

2. The `header` - This argument refers to the column headers that may (TRUE) or may not (FALSE) exist **in the plain text file you are reading in**.

- for `read.table() header = FALSE` (by default, it assumes you do not have column names)
- for `read.csv() header = TRUE` (by default, it assumes that all your columns have names listed).

3. The `row.names` - This argument refers to the rownames.

- for `read.table()` `row.names` by default assumes that your rownames are not in the first column.
- for `read.csv()` `header = TRUE` (by default, it assumes that your rownames are in the first column).

Note: this one is tricky because the default isn't listed as such in the help file.

The take-home from the “Usage” section for `read.csv()` is that it has one mandatory argument, the path to the file and filename in quotations; in our case that is `data/mouse_exp_design.csv`

7.1.4 Create a data frame by reading in the file

Let's read in the `mouse_exp_design.csv` file and create a new data frame called `metadata`.

```
metadata <- read.csv(file="data/mouse_exp_design.csv")
```

We can see if it has successfully been read in by running:

```
metadata
```

Exercise 1

1. Read “project-summary.txt” in to R using `read.table()` with the appropriate arguments and store it as the variable `proj_summary`. To figure out the appropriate arguments to use with `read.table()`, keep the following in mind:
 - all the columns in the input text file have column name/headers
 - you want the first column of the text file to be used as row names (hint: look up the input for the `row.names =` argument in `read.table()`)
 2. Display the contents of `proj_summary` in your console
-

7.2 Inspecting data structures

There are a wide selection of base functions in R that are useful for inspecting your data and summarizing it. Below is a non-exhaustive list of these functions:

The list has been divided into functions that work on all types of objects, some that work only on vectors/factors (1 dimensional objects), and others that work on data frames and matrices (2 dimensional objects).

All data structures - content display:

- `str()`: compact display of data contents (similar to what you see in the Global environment)

- `class()`: displays the data type for vectors (e.g. character, numeric, etc.) and data structure for dataframes, matrices
- `summary()`: detailed display of the contents of a given object, including descriptive statistics, frequencies
- `head()`: prints the first 6 entries (elements for 1-D objects, rows for 2-D objects)
- `tail()`: prints the last 6 entries (elements for 1-D objects, rows for 2-D objects)

Vector and factor variables:

- `length()`: returns the number of elements in a vector or factor

Dataframe and matrix variables:

- `dim()`: returns dimensions of the dataset (number_of_rows, number_of_columns) [Note, row numbers will always be displayed before column numbers in R]
- `nrow()`: returns the number of rows in the dataset
- `ncol()`: returns the number of columns in the dataset
- `rownames()`: returns the row names in the dataset
- `colnames()`: returns the column names in the dataset

Let's use the `metadata` file that we created to test out data inspection functions.

```
head(metadata)
str(metadata)
dim(metadata)
nrow(metadata)
ncol(metadata)
class(metadata)
colnames(metadata)
```

Exercise 2

- What is the class of each column in `metadata` (use one command)?
- What is the median of the replicates in `metadata` (use one command)?

Exercise 3

- Use the `class()` function on `glengths` and `metadata`, how does the output differ between the two?
- Use the `summary()` function on the `proj_summary` dataframe, what is the median “`rRNA_rate`”?
- How long is the `samplegroup` factor?
- What are the dimensions of the `proj_summary` dataframe?
- When you use the `rownames()` function on `metadata`, what is the *data structure* of the output?
- [Optional] How many elements in (how long is) the output of `colnames(proj_summary)`? Don't count, but use another function to determine this.

Chapter 8

Dataframes and matrices

8.1 Dataframes

Dataframes (and matrices) have 2 dimensions (rows and columns), so if we want to select some specific data from it we need to specify the “coordinates” we want from it. We use the same square bracket notation but rather than providing a single index, there are *two indices required*. Within the square bracket, **row numbers come first followed by column numbers (and the two are separated by a comma)**. Let’s explore the `metadata` dataframe, shown below are the first six samples:

	genotype	celltype	replicate
Sample1	Wt	typeA	1
	Wt	typeA	2
Sample3	Wt	typeA	3
	KO	typeA	1
Sample5	KO	typeA	2
	KO	typeA	3

Let's say we wanted to extract the wild type (`Wt`) value that is present in the first row and the first column. To extract it, just like with vectors, we give the name of the data frame that we want to extract from, followed by the square brackets. Now inside the square brackets we give the coordinates or indices for the rows in which the value(s) are present, followed by a comma, then the coordinates or indices for the columns in which the value(s) are present. We know the wild type value is in the first row if we count from the top, so we put a one, then a comma. The wild type value is also in the first column, counting from left to right, so we put a one in the columns space too.

```
# Extract value 'Wt'
metadata[1, 1]
```

Now let's extract the value `1` from the first row and third column.

```
# Extract value '1'
metadata[1, 3]
```

Now if you only wanted to select based on rows, you would provide the index for the rows and leave the columns index blank. The key here is to include the comma, to let R know that you are accessing a 2-dimensional data structure:

```
# Extract third row
metadata[3, ]
```

What kind of data structure does the output appear to be? We see that it is two-dimensional with row names and column names, so we can surmise that it's likely a data frame.

If you were selecting specific columns from the data frame - the rows are left blank:

```
# Extract third column
metadata[, 3]
```

What kind of data structure does this output appear to be? It looks different from the data frame, and we really just see a series of values output, indicating a vector data structure. This happens by default. R automatically drops to the simplest data structure possible. Oftentimes however, we would like to keep our single column as a data frame. To do this, there is an argument we can add when subsetting called `drop`, by changing its value to `FALSE` the output is kept as a data frame.

```
# Extract third column as a data frame
metadata[, 3, drop = FALSE]
```

Just like with vectors, you can select multiple rows and columns at a time. Within the square brackets, you need to provide a vector of the desired values.

We can extract consecutive rows or columns using the colon (`:`) to create the vector of indices to extract.

```
# Dataframe containing first two columns
metadata[, 1:2]
```

Alternatively, we can use the combine function (`c()`) to extract any number of rows or columns. Let's extract the first, third, and sixth rows.

```
# Data frame containing first, third and sixth rows
metadata[c(1,3,6), ]
```

For larger datasets, it can be tricky to remember the column number that corresponds to a particular variable. It's, therefore, often better to use column/row names to refer to extract particular values, and it makes your code easier to read and your intentions clearer.

```
# Extract the celltype column for the first three samples
metadata[c("sample1", "sample2", "sample3") , "celltype"]
```

If you need to remind yourself of the column/row names, the following functions are helpful:

```
# Check column names of metadata data frame
colnames(metadata)

# Check row names of metadata data frame
rownames(metadata)
```

If only a single column is to be extracted from a data frame, there is a useful shortcut available. If you type the name of the data frame, then the \$, you have the option to choose which column to extract. For instance, let's extract the entire genotype column from our dataset:

```
# Extract the genotype column
metadata$genotype
```

The output will always be a vector, and if desired, you can continue to treat it as a vector. For example, if we wanted the genotype information for the first five samples in `metadata`, we can use the square brackets ([]) with the indices for the values from the vector to extract:

```
# Extract the first five values/elements of the genotype column
metadata$genotype[1:5]
```

Unfortunately, there is no equivalent \$ syntax to select a row by name.

Exercises

1. Return a data frame with only the `genotype` and `replicate` column values for `sample2` and `sample8`.
 2. Return the fourth and ninth values of the `replicate` column.
 3. Extract the `replicate` column as a data frame.
-

8.1.1 Selecting using indices with logical operators

With data frames, similar to vectors, we can use logical expressions to extract the rows or columns in the data frame with specific values. First, we need to determine the indices in a rows or columns where a logical expression is TRUE, then we can extract those rows or columns from the data frame.

For example, if we want to return only those rows of the data frame with the `celltype` column having a value of `typeA`, we would perform two steps:

1. Identify which rows in the `celltype` column have a value of `typeA`.
2. Use those TRUE values to extract those rows from the data frame.

To do this we would extract the column of interest as a vector, with the first value corresponding to the first row, the second value corresponding to the second row, so on and so forth. We use that vector in the logical expression. Here we are looking for values to be equal to `typeA`, so our logical expression would be:

```
metadata$celltype == "typeA"
```

This will output TRUE and FALSE values for the values in the vector. The first six values are TRUE, while the last six are FALSE. This means the first six rows of our metadata have a value of `typeA` while the last six do not. We can save these values to a variable, which we can call whatever we would like; let's call it `logical_idx`.

```
logical_idx <- metadata$celltype == "typeA"
```

Now we can use those TRUE and FALSE values to extract the rows that correspond to the TRUE values from the metadata data frame. We will extract as we normally would a data frame with `metadata[,]`, and we need to make sure we put the `logical_idx` in the row's space, since those TRUE and FALSE values correspond to the ROWS for which the expression is TRUE/FALSE. We will leave the column's space blank to return all columns.

```
metadata[logical_idx, ]
```

8.1.2 Logical operators using the `which()` function

As you might have guessed, we can also use the `which()` function to return the indices for which the logical expression is TRUE. For example, we can find the indices where the `celltype` is `typeA` within the `metadata` data frame:

```
which(metadata$celltype == "typeA")
```

This returns the values one through six, indicating that the first 6 values or rows are true, or equal to `typeA`. We can save our indices for which rows the logical expression is true to a variable we'll call `idx`, but, again, you could call it anything you want.

```
idx <- which(metadata$celltype == "typeA")
```

Then, we can use these indices to indicate the rows that we would like to return by extracting that data as we have previously, giving the `idx` as the rows that we would like to extract, while returning all columns:

```
metadata[idx, ]
```

Let's try another subsetting. Extract the rows of the metadata data frame for only the replicates 2 and 3. First, let's create the logical expression for the column of interest (`replicate`):

```
which(metadata$replicate > 1)
```

This should return the indices for the rows in the `replicate` column within `metadata` that have a value of 2 or 3. Now, we can save those indices to a variable and use that variable to extract those corresponding rows from the `metadata` table.

```
idx <- which(metadata$replicate > 1)

metadata[idx, ]
```

Alternatively, instead of doing this in two steps, we could use nesting to perform in a single step:

```
metadata[which(metadata$replicate > 1), ]
```

Either way works, so use the method that is most intuitive for you.

So far we haven't stored as variables any of the extractions/subsettings that we have performed. Let's save this output to a variable called `sub_meta`:

```
sub_meta <- metadata[which(metadata$replicate > 1), ]
```

Exercise

Subset the `metadata` dataframe to return only the rows of data with a genotype of K0.

Chapter 9

Logical operators for matching

9.1 Logical operators to match elements

Oftentimes, we encounter different analysis tools that require multiple input datasets. It is not uncommon for these inputs to need to have the same row names, column names, or unique identifiers in the same order to perform the analysis. Therefore, knowing how to reorder datasets and determine whether the data matches is an important skill.

In our use case, we will be working with genomic data. We have gene expression data generated by RNA-seq, which we had downloaded previously; in addition, we have a metadata file corresponding to the RNA-seq samples. The metadata contains information about the samples present in the gene expression file, such as which sample group each sample belongs to and any batch or experimental variables present in the data.

Let's read in our gene expression data (RPKM matrix) that we downloaded previously:

```
rpkm_data <- read.csv("data/counts.rpkm.csv")
```

Take a look at the first few lines of the data matrix to see what's in there.

```
head(rpkm_data)
```

It looks as if the sample names (header) in our data matrix are similar to the row names of our metadata file, but it's hard to tell since they are not in the same order. We can do a quick check of the number of columns in the count data and the rows in the metadata and at least see if the numbers match up.

```
ncol(rpkm_data)  
nrow(metadata)
```

What we want to know is, **do we have data for every sample that we have metadata?**

9.2 The %in% operator

This operator is well-used and convenient once you get the hang of it. The operator is known as `exactly in` and is used with the following syntax:

```
vector1 %in% vector2
```

It will take each element from vector1 as input, one at a time, and evaluate if the element is present in vector2. The two vectors do not have to be the same size. This operation will return a vector containing logical values to indicate whether or not there is a match. The new vector will be of the same length as vector1. Take a look at the example below:

```
A <- c(1,3,5,7,9,11) # odd numbers
B <- c(2,4,6,8,10,12) # even numbers

# test to see if each of the elements of A is in B
A %in% B
```

```
## [1] FALSE FALSE FALSE FALSE FALSE FALSE
```

Since vector A contains only odd numbers and vector B contains only even numbers, the operation returns a logical vector containing six FALSE, suggesting that no element in vector A is present in vector B. Let's change a couple of numbers inside vector B to match vector A:

```
A <- c(1,3,5,7,9,11) # odd numbers
B <- c(2,4,6,8,1,5) # add some odd numbers in
```

```
# test to see if each of the elements of A is in B
A %in% B
```

```
## [1] TRUE FALSE TRUE FALSE FALSE FALSE
```

The returned logical vector denotes which elements in A are also in B - the first and third elements, which are 1 and 5.

Note: this function is not reversible; i.e. B %in% A will give a different answer.

We saw previously that we could use the output from a logical expression to subset data by returning only the values corresponding to TRUE. Therefore, we can use the output logical vector to subset our data, and return only those elements in A, which are also in B by returning only the TRUE values:

A	1	3	5	7	9	11
Index	1	2	3	4	5	6

B	2	4	6	8	1	5
Index	1	2	3	4	5	6

```
intersection <- A %in% B
intersection
```

intersection	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE
Index	1	2	3	4	5	6

A[intersection]	1	5
Index	1	2

In these previous examples, the vectors were so small that it's easy to check every logical value by eye; but this is not practical when we work with large datasets (e.g. a vector with 1000 logical values). Instead, we can use `any` function. Given a logical vector, this function will tell you whether **at least one value** is TRUE. It provides us a quick way to assess if **any of the values contained in vector A are also in vector B**:

```
any(A %in% B)
```

The `all` function is also useful. Given a logical vector, it will tell you whether **all values** are TRUE. If there is at least one FALSE value, the `all` function will return a FALSE. We can use this function to assess whether **all elements from vector A are contained in vector B**.

```
all(A %in% B)
```

Exercise 1

1. Using the A and B vectors created above, evaluate each element in B to see if there is a match in A
2. Subset the B vector to only return those values that are also in A.

Suppose we had two vectors containing same values. How can we check **if those values are in the same order in each vector**? In this case, we can use `==` operator to compare each element of the same position from two vectors. The operator returns a logical vector indicating TRUE/FALSE at each position. Then we can use `all()` function to check if all values in the returned vector are TRUE. If all values are TRUE, we know that these two vectors are the same. Unlike `%in%` operator, `==` operator requires that **two vectors are of equal length**.

```
A <- c(10,20,30,40,50)
B <- c(50,40,30,20,10) # same numbers but backwards

# test to see if each element of A is in B
A %in% B

# test to see if each element of A is in the same position in B
A == B

# use all() to check if they are a perfect match
all(A == B)
```

Let's try this on our genomic data, and see whether we have metadata information for all samples in our expression data. We'll start by creating two vectors: one is the `rownames` of the metadata, and one is the `colnames` of the RPKM data. These are base functions in R which allow you to extract the row and column names as a vector:

```
x <- rownames(metadata)
y <- colnames(rpkm_data)
```

Now check to see that all of `x` are in `y`:

```
all(x %in% y)
```

Note that we can use nested functions in place of `x` and `y` and still get the same result:

```
all(rownames(metadata) %in% colnames(rpkm_data))
```

We know that all samples are present, but are they in the same order?

```
x == y
all(x == y)
```

Looks like all of the samples are there, but they need to be reordered. To reorder our genomic samples, we will learn different ways to reorder data in our next lesson. But before that, let's work on exercise 2 to consolidate concepts from this lesson.

Exercise 2

We have a list of 6 marker genes that we are very interested in. Our goal is to extract count data for these genes using the `%in%` operator from the `rpkm_data` data frame, instead of scrolling through `rpkm_data` and finding them manually.

First, let's create a vector called `important_genes` with the Ensembl IDs of the 6 genes we are interested in:

```
important_genes <- c("ENSMUSG00000083700", "ENSMUSG00000080990",
"ENSMUSG00000065619", "ENSMUSG00000047945", "ENSMUSG00000081010",
"ENSMUSG00000030970")
```

1. Use the `%in%` operator to determine if all of these genes are present in the row names of the `rpkm_data` data frame.
2. Extract the rows from `rpkm_data` that correspond to these 6 genes using `[]` and the `%in%` operator. Double check the row names to ensure that you are extracting the correct rows.

3. **Bonus question:** Extract the rows from `rpkm_data` that correspond to these 6 genes using `[]`, but without using the `%in%` operator.
-

Chapter 10

Reordering to match datasets

10.1 Reordering data to match

In the previous lesson, we learned how to determine whether the same data is present in two datasets, in addition to, whether it is in the same order. In this lesson, we will explore how to reorder the data such that the datasets are matching.

Exercise

Now that we know how to reorder using indices, let's try to use it to reorder the contents of one vector to match the contents of another. Let's create the vectors `first` and `second` as detailed below:

<code>first</code>	A	B	C	D	E
<code>Index</code>	1	2	3	4	5

<code>second</code>	B	D	E	A	C
<code>Index</code>	1	2	3	4	5

```
first <- c("A", "B", "C", "D", "E")
second <- c("B", "D", "E", "A", "C") # same letters but different order
```

How would you reorder the `second` vector to match `first`?

If we had large datasets, it would be difficult to reorder them by searching for the indices of the matching elements, and it would be quite easy to make a typo or mistake. To help with matching datasets, there is a function called `match()`.

10.2 The `match` function

We can use the `match()` function to match the values in two vectors. We'll be using it to evaluate which values are present in both vectors, and how to reorder the elements to make the values match.

`match()` takes 2 arguments. The first argument is a vector of values in the order you want, while the second argument is the vector of values to be reordered such that it will match the first:

1. a vector of values in the order you want
2. a vector of values to be reordered

The function returns the position of the matches (indices) with respect to the second vector, which can be used to re-order it so that it matches the order in the first vector. Let's use `match()` on the first and second vectors we created.

```
match(first,second)
[1] 4 1 5 2 3
```

The output is the indices for how to reorder the second vector to match the first. *These indices match the indices that we derived manually before.*

Now, we can just use the indices to reorder the elements of the `second` vector to be in the same positions as the matching elements in the `first` vector:

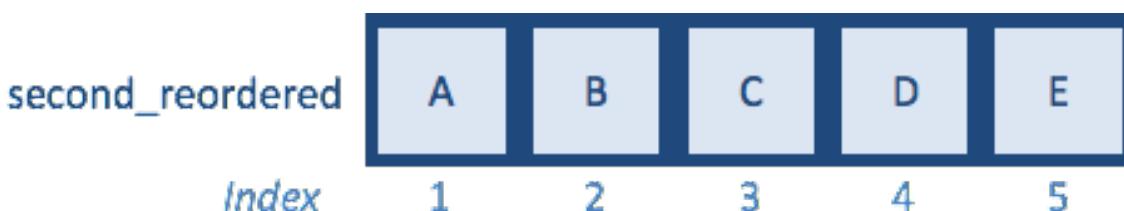
```
# Saving indices for how to reorder `second` to match `first`
reorder_idx <- match(first,second)
```

Then, we can use those indices to reorder the second vector similar to how we ordered with the manually derived indices.

```
# Reordering the second vector to match the order of the first vector
second[reorder_idx]
```

If the output looks good, we can save the reordered vector to a new variable.

```
# Reordering and saving the output to a variable
second_reordered <- second[reorder_idx]
```



Now that we know how `match()` works, let's change vector `second` so that only a subset are retained:

```
first <- c("A", "B", "C", "D", "E")
second <- c("D", "B", "A") # remove values
```

first	A	B	C	D	E
<i>Index</i>	1	2	3	4	5
second	D	B	A		
<i>Index</i>	1	2	3		

And try to `match()` again:

```
match(first, second)
[1] 3 2 NA 1 NA
```

We see that the `match()` function takes every element in the first vector and finds the position of that element in the second vector, and if that element is not present, will return a missing value of NA. The value NA represents missing data for any data type within R. In this case, we can see that the `match()` function output represents the value at position 3 as first, which is A, then position 2 is next, which is B, the value coming next is supposed to be C, but it is not present in the `second` vector, so NA is returned, so on and so forth.

NOTE: For values that don't match by default return an NA value. You can specify what values you would have it assigned using `nomatch` argument. Also, if there is more than one matching value found only the first is reported.

If we rearrange `second` using these indices, then we should see that all the values present in both vectors are in the same positions and NAs are present for any missing values.

```
second[match(first, second)]
```

Reordering genomic data using `match()` function

While the input to the `match()` function is always going to be to vectors, often we need to use these vectors to reorder the rows or columns of a data frame to match the rows or columns of another data frame. Let's explore how to do this with our use case featuring RNA-seq data. To perform differential gene expression analysis, we have a data frame with the expression data or counts for every sample and another data frame with the information about to which condition each sample belongs.

For the tools doing the analysis, the samples in the counts data, which are the column names, need to be the same and in the same order as the samples in the metadata data frame, which are the rownames.

We can take a look at these samples in each dataset by using the `rownames()` and `colnames()` functions.

```
# Check row names of the metadata
rownames(metadata)

# Check the column names of the counts data
colnames(rpkm_data)
```

We see the row names of the metadata are in a nice order starting at `sample1` and ending at `sample12`, while the column names of the counts data look to be the same samples, but are randomly ordered. Therefore, we want to reorder the columns of the counts data to match the order of the row names of the metadata. To do so, we will use the `match()` function to match the row names of our metadata with the column names of our counts data, so these will be the arguments for `match`.

To do so, we will use the `match` function to match the row names of our metadata with the column names of our counts data, so these will be the arguments for `match()`.

Within the `match()` function, the `rownames` of the metadata is the vector in the order that we want, so this will be the first argument, while the column names of the count or `rpkm` data is the vector to be reordered. We will save these indices for how to reorder the column names of the count data such that it matches the `rownames` of the metadata to a variable called `genomic_idx`.

```
genomic_idx <- match(rownames(metadata), colnames(rpkm_data))
genomic_idx
```

The `genomic_idx` represents how to re-order the column names in our counts data to be identical to the row names in metadata.

Now we can create a new counts data frame in which the columns are re-ordered based on the `match()` indices. Remember that to reorder the rows or columns in a data frame we give the name of the data frame followed by square brackets, and then the indices for how to reorder the rows or columns.

Our `genomic_idx` represents how we would need to reorder the `columns` of our count data such that the column names would be in the same order as the row names of our metadata. Therefore, we need to add our `genomic_idx` to the `columns position`. We are going to save the output of the reordering to a new data frame called `rpkm_ordered`.

```
# Reorder the counts data frame to have the sample names in the same order as the metadata data frame
rpkm_ordered <- rpkm_data[, genomic_idx]
```

Check and see what happened by clicking on the `rpkm_ordered` in the Environment window or using the `View()` function.

```
# View the reordered counts
View(rpkm_ordered)
```

We can see the sample names are now in a nice order from sample 1 to 12, just like the metadata.

You can also verify that column names of this new data matrix matches the metadata row names by using the `all` function:

```
all(rownames(metadata) == colnames(rpkm_ordered))
```

Now that our samples are ordered the same in our metadata and counts data, if these were raw counts (not RPKM) we could proceed to perform differential expression analysis with this dataset.

Exercises

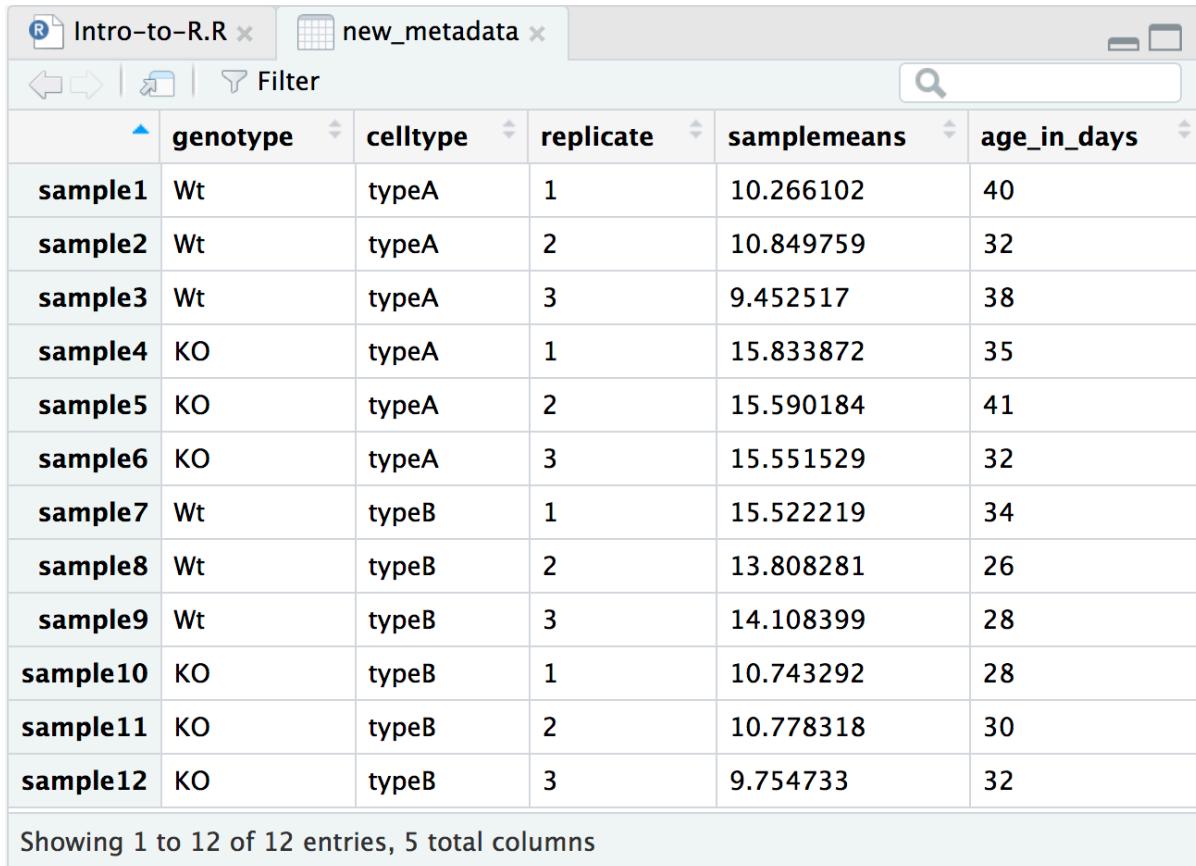
1. After talking with your collaborator, it becomes clear that `sample2` and `sample9` were actually from a different mouse background than the other samples and should not be part of our analysis. Create a new variable called `subset_rpkm` that has these columns removed from the `rpkm_ordered` data frame.
 2. Use the `match()` function to subset the `metadata` data frame so that the row names of the `metadata` data frame match the column names of the `subset_rpkm` data frame.
-

Chapter 11

Plotting and data visualization

11.1 Dataframe setup for visualization

In this lesson we want to make plots to evaluate the average expression in each sample and its relationship with the age of the mouse. So, to this end, we will be adding a couple of additional columns of information to the `metadata` data frame that we can utilize for plotting.



The screenshot shows the RStudio interface with the 'new_metadata' data frame open. The data frame has 12 rows and 5 columns. The columns are: genotype, celltype, replicate, samplemeans, and age_in_days. The data is as follows:

	genotype	celltype	replicate	samplemeans	age_in_days
sample1	Wt	typeA	1	10.266102	40
sample2	Wt	typeA	2	10.849759	32
sample3	Wt	typeA	3	9.452517	38
sample4	KO	typeA	1	15.833872	35
sample5	KO	typeA	2	15.590184	41
sample6	KO	typeA	3	15.551529	32
sample7	Wt	typeB	1	15.522219	34
sample8	Wt	typeB	2	13.808281	26
sample9	Wt	typeB	3	14.108399	28
sample10	KO	typeB	1	10.743292	28
sample11	KO	typeB	2	10.778318	30
sample12	KO	typeB	3	9.754733	32

Showing 1 to 12 of 12 entries, 5 total columns

11.1.1 Calculating average expression

Let's take a closer look at our counts data (`rpkm_ordered`). Each column represents a sample in our experiment, and each sample has $> 36,000$ total counts. We want to compute **the average value of expression** for each sample. Taking this one step at a time, if we just wanted the average expression for Sample 1 we can use the R base function `mean()`:

```
mean(rpkm_ordered$sample1)
```

That is great, but we need to get this information from all 12 samples, so all 12 columns. We want a vector of 12 values that we can add to the metadata data frame. What is the best way to do this?

To get the mean of all the samples in a single line of code the `map()` family of function is a good option.

11.1.2 The `map` family of functions

The `map()` family of functions is available from the `purrr` package, which is part of the tidyverse suite of packages. We can `map()` functions to execute some task/function on every element in a vector, or every column in a data frame, or every component of a list, and so on.

- `map()` creates a list.
- `map_lgl()` creates a logical vector.
- `map_int()` creates an integer vector.
- `map_dbl()` creates a “double” or numeric vector.
- `map_chr()` creates a character vector.

The syntax for the `map()` family of functions is:

```
## DO NOT RUN
map(object, function_to_apply)
```

To obtain **mean values for all samples** we can use the `map_dbl()` function which generates a numeric vector.

```
library(purrr) # Load the purrr
samplemeans <- map_dbl(rpkm_ordered, mean)
```

The output of `map_dbl()` is a *named* vector of length 12.

11.1.3 Adding data to metadata

Before we add `samplemeans` as a new column to metadata, let's create a vector with the ages of each of the mice in our data set.

```
# Create a numeric vector with ages. Note that there are 12 elements here
age_in_days <- c(40, 32, 38, 35, 41, 32, 34, 26, 28, 28, 30, 32)
```

Now, we are ready to combine the `metadata` data frame with the 2 new vectors to create a new data frame with 5 columns:

```
# Add the new vectors as the last columns to the metadata  
  
new_metadata <- data.frame(metadata, samplemeans, age_in_days)  
  
# Take a look at the new_metadata object  
head(new_metadata)
```

Using new_metadata, we are now ready for plotting and data visualization.

Chapter 12

Plotting with ggplot2

12.1 Data Visualization with ggplot2

This chapter will teach you how to visualise your data using ggplot2. R has several systems for making graphs, but ggplot2 is one of the most elegant and most versatile. ggplot2 implements the grammar of graphics, a coherent system for describing and building graphs. With ggplot2, you can do more faster by learning one system and applying it in many places.

For this lesson, you will need the `new_metadata` data frame. Load it into your environment as follows:

```
## load the new_metadata data frame into your environment from a .RData object
load("data/new_metadata.RData")
```

Next, let's check if it was successfully loaded into the environment:

```
# this data frame should have 12 rows and 5 columns
head(new_metadata)
```

Great, we are now ready to move forward!

The ggplot2 syntax takes some getting used to, but once you get it, you will find it's extremely powerful and flexible. We will start with drawing a simple x-y scatterplot of `samplemeans` versus `age_in_days` from the `new_metadata` data frame. Note that ggplot2 expects a “data frame”.

Let's start by loading the ggplot2 library:

```
library(ggplot2)
```

The `ggplot()` function is used to **initialize the basic graph structure**, then we add to it. The basic idea is that you specify different parts of the plot using additional functions one after the other and combine them using the `+` operator; the functions in the resulting code chunk are called layers.

Let's start:

```
ggplot(new_metadata) # what happens?
```

You get a blank plot, because you need to **specify additional layers** using the `+` operator.

The **geom (geometric) object** is the layer that specifies what kind of plot we want to draw. A plot **must have at least one geom**; there is no upper limit. Examples include:

- points (`geom_point`, `geom_jitter` for scatter plots, dot plots, etc)
- lines (`geom_line`, for time series, trend lines, etc)
- boxplot (`geom_boxplot`, for, well, boxplots!)

Let's add a "geom" layer to our plot using the `+` operator, and since we want a scatter plot so we will use `geom_point()`.

```
ggplot(new_metadata) +
  geom_point() # note what happens here
```

Why do we get an error?

We get an error because each type of `geom` usually has a **required set of aesthetics** to be set. "Aesthetics" are set with the `aes()` function can be set nested within `geom_point()` or within `ggplot()`.

The minimal `ggplot` function requires the following arguments:

```
# Minimal ggplot template:
ggplot(<DATA>) +          # 1. specify data set to use
<GEOM_function>(
  aes(<MAPPING>)).        # 2. specify geom
                           # 3. map x and y axes
```

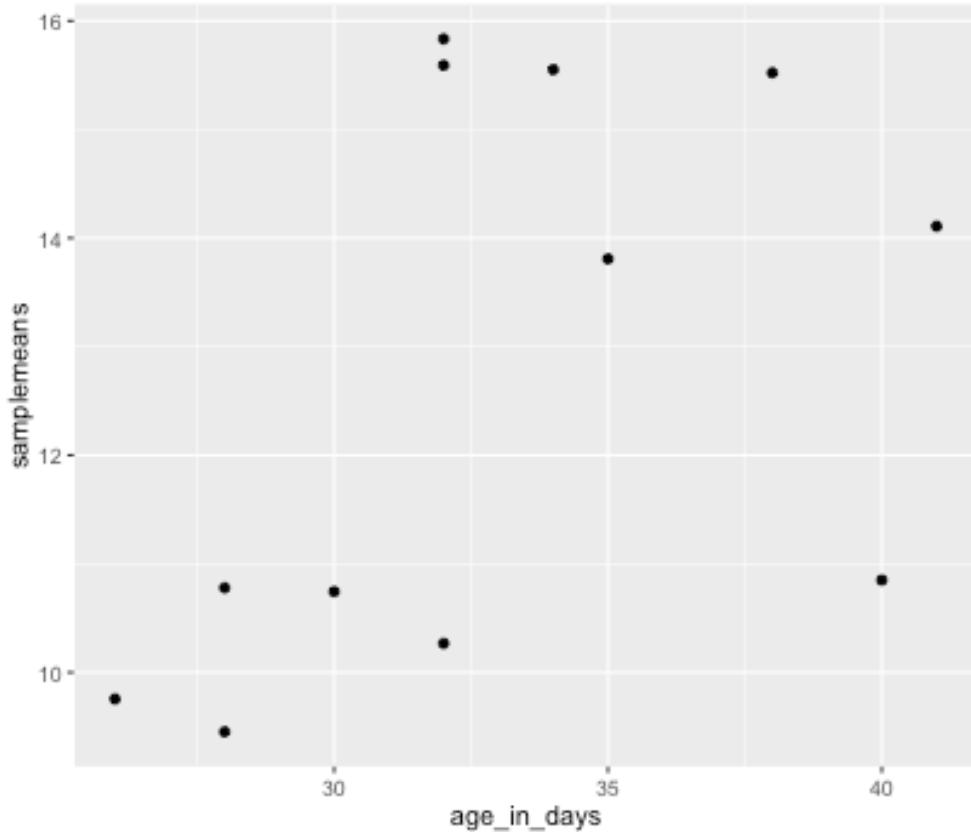
12.1.1 Aesthetics

The `aes()` function has many different arguments, and all of those arguments take columns from the original data frame as input. It can be used to specify many plot elements including the following:

- position (i.e., on the x and y axes)
- color ("outside" color)
- fill ("inside" color)
- shape (of points)
- etc.

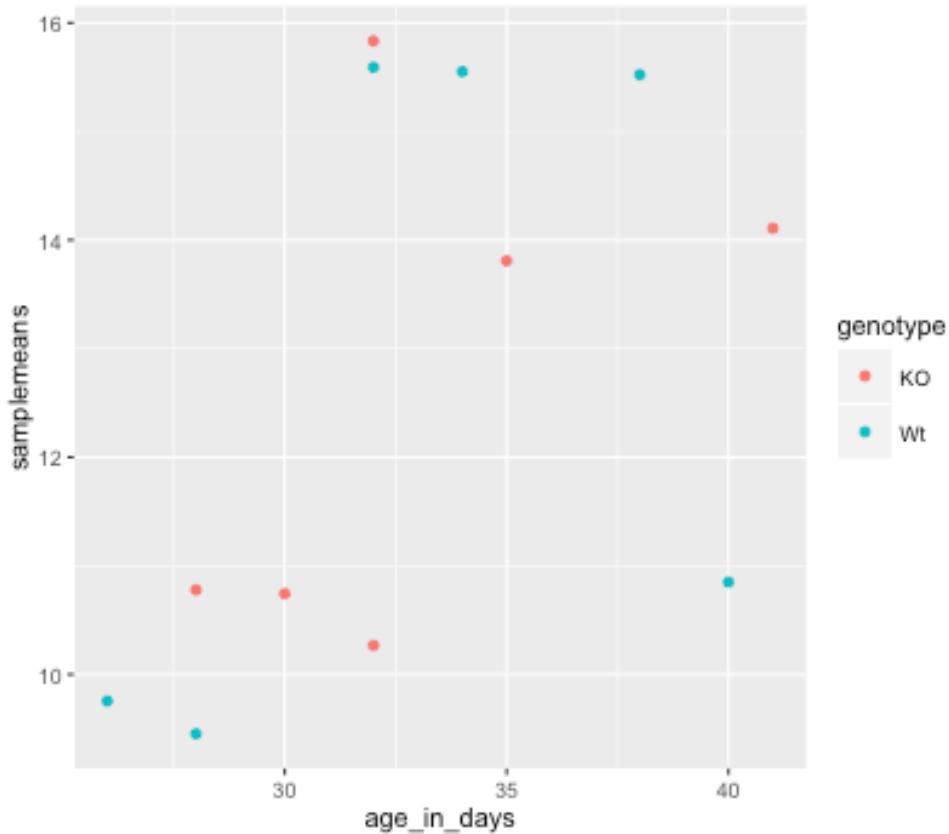
To start, we will specify x- and y-axis since `geom_point` requires the most basic information about a scatterplot, i.e. what you want to plot on the x and y axes. All of the other plot elements mentioned above are optional.

```
ggplot(new_metadata) +
  geom_point(aes(x = age_in_days, y= samplemeans))
```



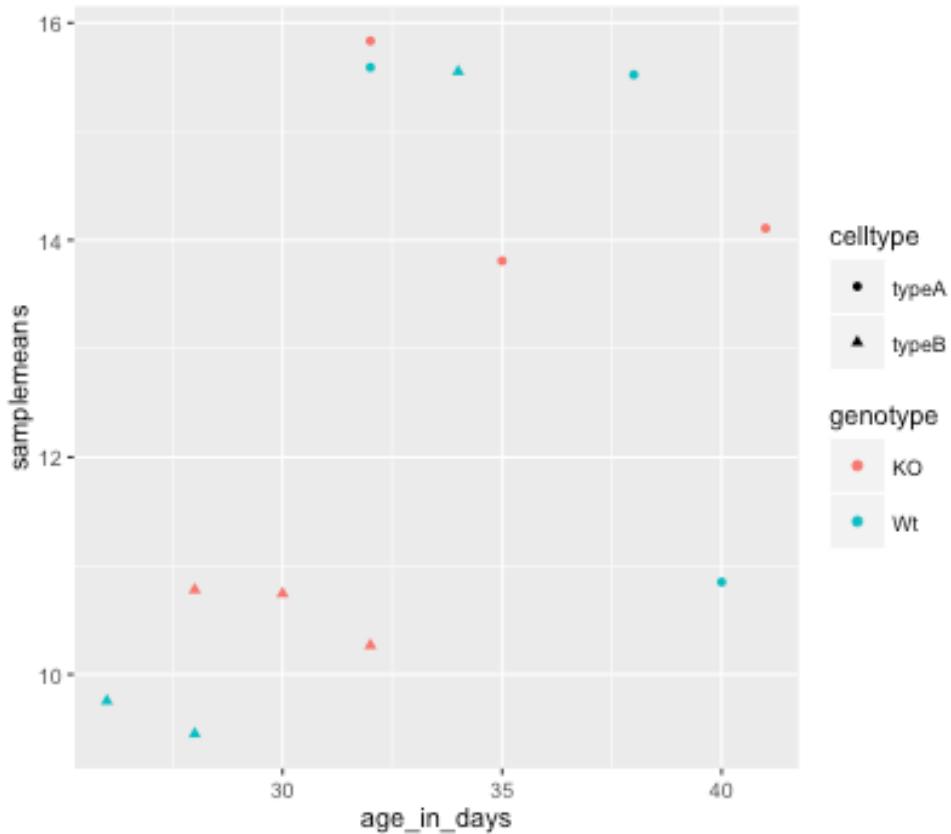
Now that we have the required aesthetics, let's add some extras like color to the plot. We can **color the points on the plot based on the genotype column within aes()**.

```
ggplot(new_metadata) +  
  geom_point(aes(x = age_in_days, y= samplemeans,  
                 color = genotype))
```



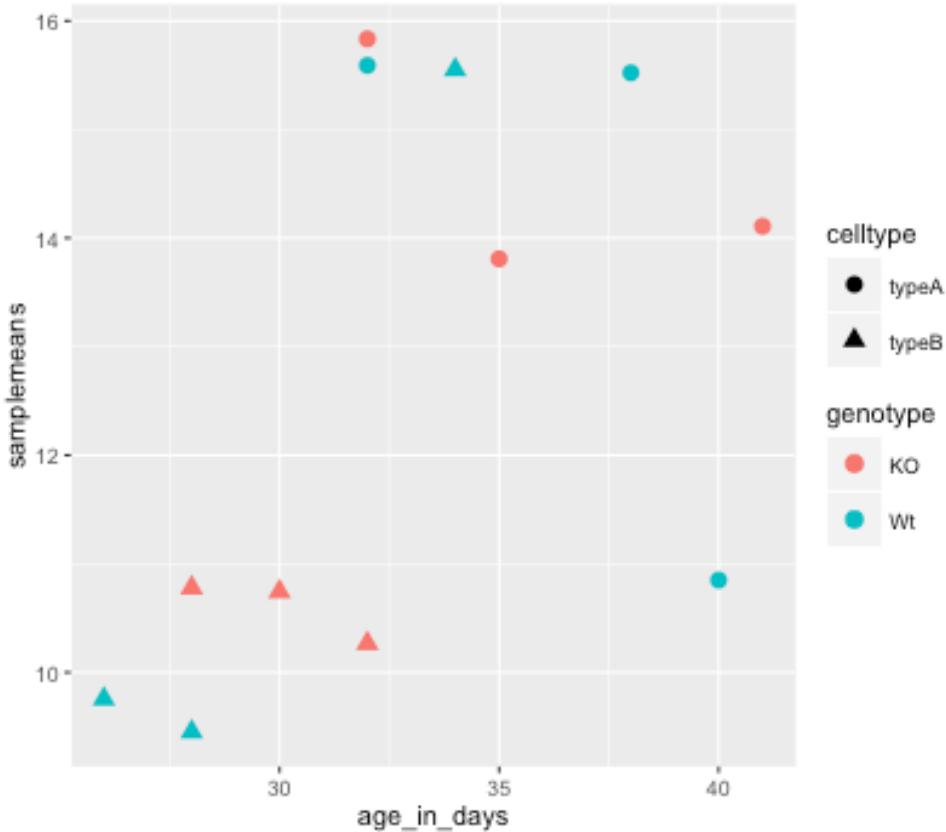
Let's try to have both **celltype** and **genotype** represented on the plot. To do this we can assign the **shape** argument in **aes()** the celltype column, so that each celltype is plotted with a different shaped data point.

```
ggplot(new_metadata) +  
  geom_point(aes(x = age_in_days, y= samplemeans,  
  color = genotype, shape=celltype))
```



The data points are quite small. We can adjust the **size of the data points** within the `geom_point()` layer, but it should **not be within `aes()`** since we are not mapping it to a column in the input data frame, instead we are just specifying a number.

```
ggplot(new_metadata) +  
  geom_point(aes(x = age_in_days, y= samplemeans,  
                 color = genotype, shape=celltype), size=2.25)
```



12.1.2 Layers

The labels on the x- and y-axis are also quite small and hard to read. To change their size, we need to add an additional `theme` layer. The `ggplot2` `theme` system handles non-data plot elements such as:

- Axis label aesthetics
- Plot background
- Facet label background
- Legend appearance

There are built-in themes we can use (i.e. `theme_bw()`) that mostly change the background/foreground colours, by adding it as additional layer. Or we can adjust specific elements of the current default theme by adding the `theme()` layer and passing in arguments for the things we wish to change.

Let's add a layer `theme_bw()`.

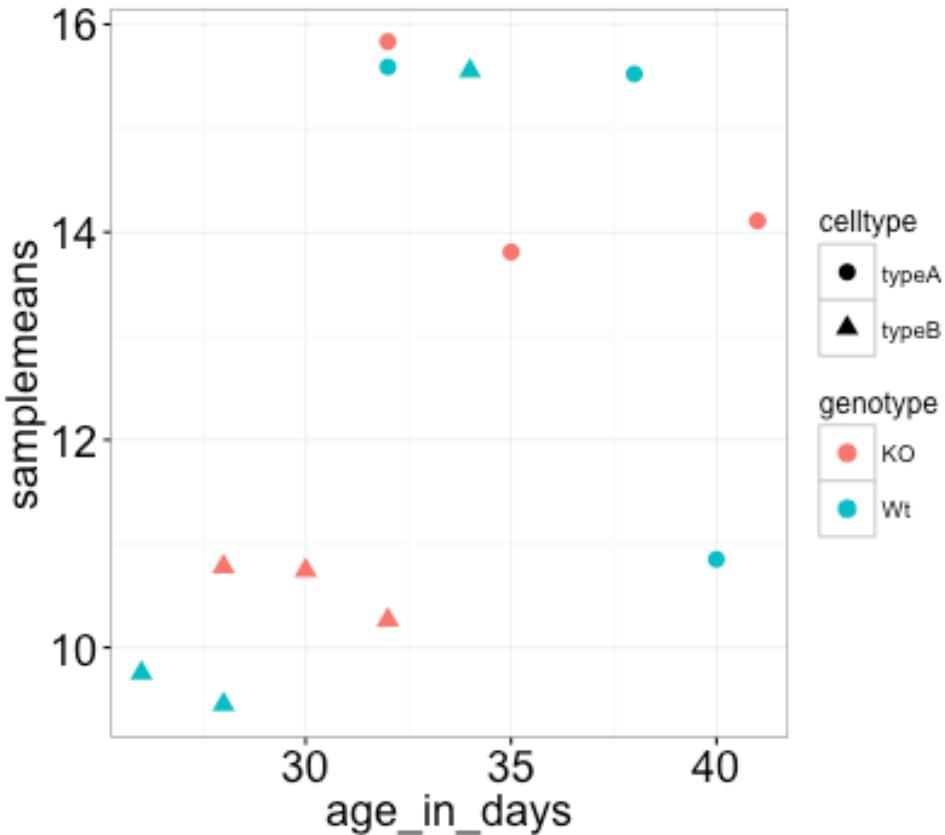
```
ggplot(new_metadata) +
  geom_point(aes(x = age_in_days, y= samplemeans,
                 color = genotype, shape=celltype), size=3.0) +
  theme_bw()
```

Do the axis labels or the tick labels get any larger by changing themes?

No, they don't. But, we can add arguments using `theme()` to change the size of axis labels ourselves. Since we will be adding this layer "on top", or after `theme_bw()`, any features we change will override what is set by the `theme_bw()` layer.

Let's increase the size of both the axes titles to be 1.5 times the default size. When modifying the size of text the `rel()` function is commonly used to specify a change relative to the default.

```
ggplot(new_metadata) +
  geom_point(aes(x = age_in_days, y= samplemeans,
                 color = genotype, shape=celltype), size=2.25) +
  theme_bw() +
  theme(axis.title = element_text(size=rel(1.5)))
```



NOTE: You can use the `example("geom_point")` function here to explore a multitude of different aesthetics and layers that can be added to your plot.

NOTE: RStudio provide this very useful cheatsheet for plotting using ggplot2. Different example plots are provided and the associated code (i.e which `geom` or `theme` to use in the appropriate situation.) We also encourage you to peruse through this useful online reference for working with ggplot2.

Exercise

1. The current axis label text defaults to what we gave as input to `geom_point` (i.e the column headers). We can change this by **adding additional layers** called `xlab()` and `ylab()` for the x- and y-axis, respectively. Add these layers to the current plot such that the x-axis is labeled "Age (days)" and the y-axis is labeled "Mean expression".

2. Use the `ggtile` layer to add a plot title of your choice.
3. Add the following new layer to the code chunk `theme(plot.title=element_text(hjust=0.5))`.

- What does it change?
- How many `theme()` layers can be added to a ggplot code chunk, in your estimation? ***

Chapter 13

Boxplot with ggplot2: exercise

13.1 Generating a Boxplot with ggplot2

A boxplot provides a graphical view of the distribution of data based on a five number summary:

- The top and bottom of the box represent the (1) first and (2) third quartiles (25th and 75th percentiles, respectively), also known as Q1 and Q3.
- The line inside the box represents the (3) median (50th percentile).
- The whiskers extending above and below the box represent the (4) maximum, and (5) minimum of a data set.
- The whiskers of the plot reach the minimum and maximum values that are not outliers.

In this case, **outliers** are determined using the interquartile range (IQR), which is defined as: $Q3 - Q1$. Any values that exceeds $1.5 \times IQR$ below Q1 or above Q3 are considered outliers and are represented as points above or below the whiskers.

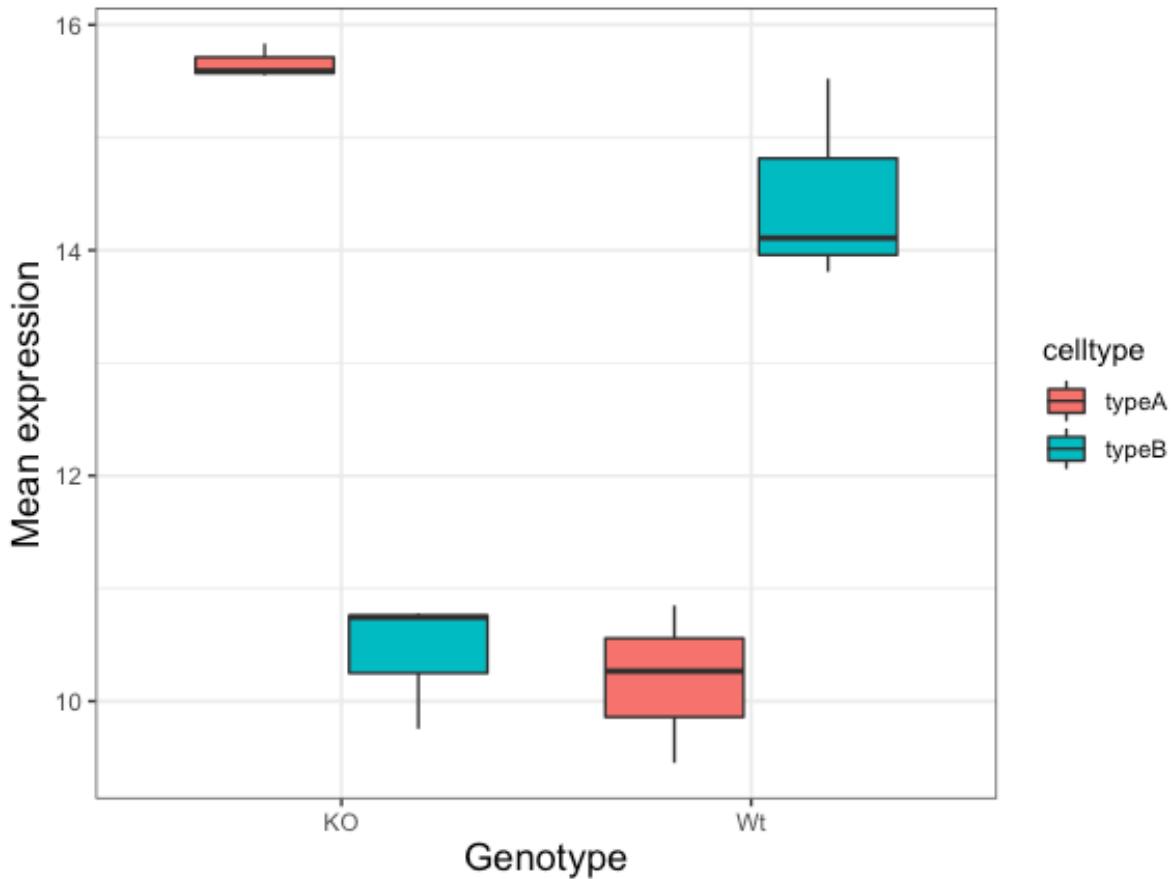
13.1.1 1. Boxplot!

Generate a boxplot using the data in the new_metadata dataframe. Create a ggplot2 code chunk with the following instructions:

- Use the `geom_boxplot()` layer to plot the differences in sample means between the Wt and KO genotypes.
- Use the `fill` aesthetic to look at differences in sample means between the celltypes within each genotype.
- Add a title to your plot.
- Add labels, ‘Genotype’ for the x-axis and ‘Mean expression’ for the y-axis.
- Make the following `theme()` changes:
 - Use the `theme_bw()` function to make the background white.
 - Change the size of your axes labels to 1.25x larger than the default.
 - Change the size of your plot title to 1.5x larger than default.
 - Center the plot title.

After running the above code the boxplot should look something like that provided below.

Genotype differences in average gene expression



13.1.2 2. Change genotype order

Let's say you wanted to have the "Wt" boxplots displayed first on the left side, and "KO" on the right. How might you go about doing this?

To do this, your first question should be - *How does ggplot2 determine what to place where on the X-axis?* * The order of the genotype on the X axis is in alphabetical order. * To change it, you need to make sure that the genotype column is a factor * And, the factor levels for that column are in the order you want on the X-axis

- Factor the `new_metadata$genotype` column without creating any extra variables/objects and change the levels to `c("Wt", "KO")` #### 3. Re-run the boxplot code chunk you created for the "Boxplot!" exercise above.

13.1.3 4. Changing default colors

You can color the boxplot differently by using some specific layers:

- Add a new layer `scale_color_manual(values=c("purple", "orange"))`.

- Do you observe a change?
- Replace `scale_color_manual(values=c("purple","orange"))` with `scale_fill_manual(values=c("purple","orange"))`.
 - Do you observe a change?
 - In the scatterplot we drew in class, add a new layer `scale_color_manual(values=c("purple","orange"))`, do you observe a difference?
 - * What do you think is the difference between `scale_color_manual()` and `scale_fill_manual()`?
- Back in your boxplot code, change the colors in the `scale_fill_manual()` layer to be your 2 favorite colors.
 - Are there any colors that you tried that did not work?

We have a separate lesson about using color palettes from the package RColorBrewer, if you are interested.

You are not restricted to using colors by writing them out as character vectors. You have the choice of a lot of colors in R, and you can do so by using their *hexadecimal code*. For example, “#FF0000” would be red and “#00FF00” would be green similarly, “#FFFFFF” would be white and “#000000” would be black. [click here](#) for more information about color palettes in R.

OPTIONAL Exercise:

13.1.4 5. Find the hexadecimal code for your 2 favourite colors (from exercise 3 above) and replace the color names with the hexadecimal codes within the ggplot2 code chunk.

Chapter 14

Saving data and plots to file

14.1 Writing data to file

Everything we have done so far has only modified the data in R; the files have remained unchanged. Whenever we want to save our datasets to file, we need to use a `write` function in R.

To write our matrix to file in comma separated format (.csv), we can use the `write.csv` function. There are two required arguments:

- the variable name of the data structure you are exporting
- the path and filename that you are exporting to.

By default the delimiter or column separator is set, and columns will be separated by a comma.

```
# Save a data frame to file
write.csv(sub_meta, file="data/subset_meta.csv")
```

Another commonly used function is `write.table`, which allows you to specify the delimiter or separator you wish to use. This function is commonly used to create tab-delimited files.

14.2 Exporting figures to file

There are two ways in which figures and plots can be output to a file (rather than simply displaying on screen).

- (1) The first (and easiest) is to export directly from the RStudio ‘Plots’ panel, by clicking on `Export` when the image is plotted. This will give you the option of `png` or `pdf` and selecting the directory to which you wish to save it to. It will also give you options to dictate the size and resolution of the output image.
- (2) The second option is to use R functions that can be hard-coded in to your script. This would allow you to run the script from start to finish and automate the process (not requiring human point-and-click actions to save). If we wanted to print our scatterplot to a pdf file format, we would need to use the functions which specifies the graphical format you intend on creating i.e. `pdf()`, `png()`, `tiff()` etc. Within the function you will need to specify a name for your image, and the width and height (optional). This will open up the device that you wish to write to:

```
## Open device for writing  
pdf("figures/scatterplot.pdf")
```

```
## Make a plot which will be written to the open device, in this case the temp file created by pdf() or png()  
  
ggplot(new_metadata) +  
  geom_point(aes(x = age_in_days, y= samplemeans, color = genotype,  
                 shape=celltype), size=rel(3.0))
```

Finally, close the “device”, or file, using the `dev.off()` function. There are also `bmp`, `tiff`, and `jpeg` functions, though the `jpeg` function has proven less stable than the others.

```
## Closing the device is essential to save the temporary file created by pdf() or png()  
dev.off()
```

Note 1: In the case of `pdf()`, if you had made additional plots before closing the device, they will all be stored in the same file with each plot usually getting its own page, unless otherwise specified.

Chapter 15

Finding help

15.1 Asking for help

The key to getting help from someone is for them to grasp your problem rapidly. You should make it as easy as possible to pinpoint where the issue might be.

1. Try to **use the correct words** to describe your problem. For instance, a package is not the same thing as a library. Most people will understand what you meant, but others have really strong feelings about the difference in meaning. The key point is that it can make things confusing for people trying to help you. **Be as precise as possible when describing your problem.**
2. **Always include the output of `sessionInfo()`** as it provides critical information about your platform, the versions of R and the packages that you are using, and other information that can be very helpful to understand your problem.

```
sessionInfo() #This time it is not interchangeable with search()
```

3. If possible, **reproduce the problem using a very small `data.frame`** instead of your 50,000 rows and 10,000 columns one, provide the small one with the description of your problem. When appropriate, try to generalize what you are doing so even people who are not in your field can understand the question.
 - To share an object with someone else, you can provide either the raw file (i.e., your CSV file) with your script up to the point of the error (and after removing everything that is not relevant to your issue). Alternatively, in particular if your question is not related to a `data.frame`, you can save any other R data structure that you have in your environment to a file:

```
# DO NOT RUN THIS!  
  
save(iris, file="/tmp/iris.RData")
```

The content of this `.RData` file is not human readable and cannot be posted directly on stackoverflow. It can, however, be emailed to someone who can read it with this command:

```
# DO NOT RUN THIS!  
  
load(file("~/Downloads/iris.RData"))
```

15.1.1 Where to ask for help?

- **Google** is often your best friend for finding answers to specific questions regarding R.
 - Cryptic error messages are very common in R - it is very likely that someone else has encountered this problem already! Start by googling the error message. However, this doesn't always work because often, package developers rely on the error catching provided by R. You end up with general error messages that might not be very helpful to diagnose a problem (e.g. “subscript out of bounds”).
- **Stackoverflow:** Search using the [r] tag. Most questions have already been answered, but the challenge is to use the right words in the search to find the answers: <http://stackoverflow.com/questions/tagged/r>. If your question hasn't been answered before and is well crafted, chances are you will get an answer in less than 5 min.
- **Your friendly colleagues:** if you know someone with more experience than you, they might be able and willing to help you.
- **The R-help:** it is read by a lot of people (including most of the R core team), a lot of people post to it, but the tone can be pretty dry, and it is not always very welcoming to new users. If your question is valid, you are likely to get an answer very fast but don't expect that it will come with smiley faces. Also, here more than everywhere else, be sure to use correct vocabulary (otherwise you might get an answer pointing to the misuse of your words rather than answering your question). You will also have more success if your question is about a base function rather than a specific package.
- **The Bioconductor support site.** This is very useful and if you tag your post, there is a high likelihood of getting an answer from the developer.
- If your question is about a specific package, see if there is a mailing list for it. Usually it's included in the DESCRIPTION file of the package that can be accessed using `packageDescription("name-of-package")`. You may also want to try to **email the author** of the package directly.
- There are also some **topic-specific mailing lists** (GIS, phylogenetics, etc...), the complete list is [here](#).

15.1.2 More resources

- The Posting Guide for the R mailing lists.
 - How to ask for R help useful guidelines
 - The Introduction to R can also be dense for people with little programming experience but it is a good place to understand the underpinnings of the R language.
 - The R FAQ is dense and technical but it is full of useful information.
-

Exercises

1. Run the following code chunks and fix all of the errors. (Note: The code chunks are independent from one another.)

```
# Create vector of work days
work_days <- c(Monday, Tuesday, Wednesday, Thursday, Friday)
```

```
# Create a function to round the output of the sum function
round_the_sum <- function(x){
  return(round(sum(x)))
}
```

```
# Create a function to add together three numbers
add_numbers <- function(x,y,z){
  sum(x,y,z)
}

add_numbers(5,9)
```

2. You try to install a package and you get the following error message:

```
Error: package or namespace load failed for 'Seurat' in loadNamespace(j <- i[[1L]], c(lib.loc, .libPaths()),  
versionCheck = vI[[j]]): there is no package called 'multtest'
```

What would you do to remedy the error?

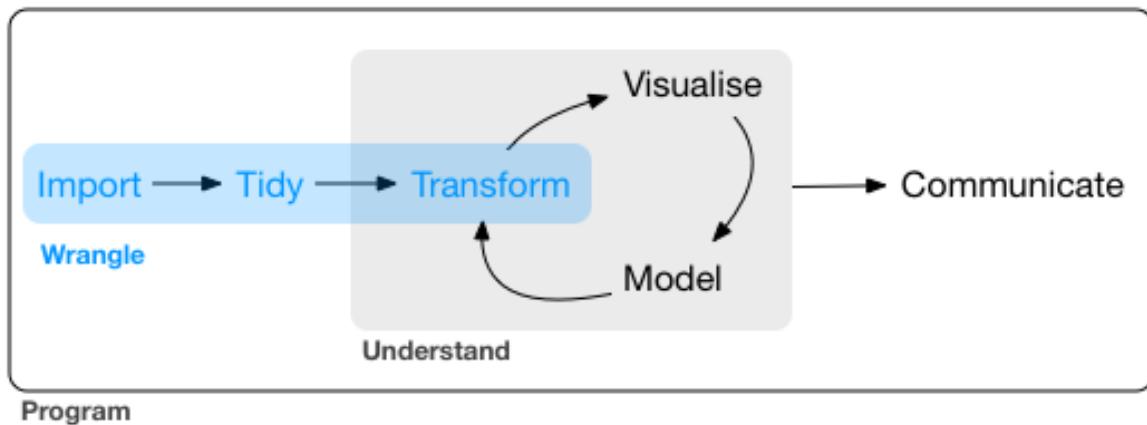
3. You would like to ask for help on an online forum. To do this you want the users of the forum to reproduce your problem, so you want to provide them as much relevant information and data as possible.

- You want to provide them with the list of packages that you currently have loaded, the version of R, your OS and package versions. Use the appropriate function(s) to obtain this information.
 - You want to also provide a small data frame that reproduces the error (if working with a large data frame, you'll need to subset it down to something small). For this exercise use the data frame `df`, and save it as an RData object called `df.RData`.
 - What code should the people looking at your help request should use to read in `df.RData`?
-

Chapter 16

Tidyverse data wrangling

An incredibly important skill of a data scientist is to be able to take data from an untidy format and get it into a tidy format. This process is often referred to as data wrangling. Generally, data wranglings skills are those that allow you to wrangle data from the format they're currently in into the tidy format you actually want them in.



Tidyverse is a suite of packages that are incredibly useful for working with data. are designed to work together to make common data science operations more user friendly. The packages have functions for data wrangling, tidying, reading/writing, parsing, and visualizing, among others.

The tidyverse

Components



The tidyverse is a collection of R packages that share common philosophies and are designed to work together. This site is a work-in-progress guide to the tidyverse and its packages.

16.1 Tidyverse basics

The Tidyverse suite of packages introduces users to a set of data structures, functions and operators to make working with data more intuitive, but is slightly different from the way we do things in base R.

Two important new concepts in tidyverse that we will focus on are pipes and tibbles.

Before we get started with pipes or tibbles, let's load the library:

```
library(tidyverse)
```

16.1.1 Pipes

To make R code more human readable, the Tidyverse tools use the pipe, `%>%`, which is part of the `dplyr` package that is installed automatically with Tidyverse. **The pipe allows the output of a previous command to be used as input to another command instead of using nested functions.**

NOTE: Shortcut to write the pipe is shift + command + M in MacOS; for Windows shift + ctrl + M

An example of using the pipe to run multiple commands:

```
## A single command
sqrt(83)
```

16.2 [1] 9.110434

```
## Base R method of running more than one command
round(sqrt(83), digits = 2)
```

16.3 [1] 9.11

```
## Running more than one command with piping
sqrt(83) %>%
  round(digits = 2)
```

16.4 [1] 9.11

The pipe represents a much easier way of writing and deciphering R code.

Exercises

1. Create a vector of random numbers using the code below:

```
random_numbers <- c(81, 90, 65, 43, 71, 29)
```

2. Use the pipe (%>%) to perform two steps in a single line:

- Take the mean of `random_numbers` using the `mean()` function.
 - Round the output to three digits using the `round()` function.
-

16.4.1 Tibbles

A core component of tidyverse is the tibble. **Tibbles are data frames, but have several properties that make it superior. For example, printing the tibble to screen displays the data types in each of the columns and the dimensions. Another very handy feature of tibbles is that by default they will only print out the first 10 rows and as many columns as fit in your window. This is important when you are working with large datasets – as we are.

Tibbles can be created directly using the `tibble()` function or data frames can be converted into tibbles using `as_tibble(name_of_df)`. It is also easy to convert `tibbles` into dataframes with the `as.data.frame()` function.

NOTE: The function `as_tibble()` will ignore row names, so if a column representing the row names is needed, then the function `rownames_to_column(name_of_df)` should be run prior to turning the `data.frame` into a tibble. `rownames_to_column()` takes the rownames and adds it as a column in the data frame.

16.5 Experimental data

We're going to explore the Tidyverse suite of tools to wrangle our data to prepare it for visualization. You should have `gprofiler_results_Mov10oe.tsv` in your R project's `data` folder earlier.

The dataset:

- Represents the **functional analysis results**, including the biological processes, functions, pathways, or conditions that are over-represented in a given list of genes.
- Our gene list was generated by **differential gene expression analysis** and the genes represent differences between **control mice** and **mice over-expressing a gene involved in RNA splicing**.

The functional analysis that we will focus on involves **gene ontology (GO) terms**, which:

- describe the roles of genes and gene products
- organized into three controlled vocabularies/ontologies (domains):
 - biological processes (BP) (e.g. DNA repair, metabolism)
 - molecular functions (MF) (e.g. kinase, protease)
 - cellular components (CC) (where they work e.g. nucleus, mitochondria)

This shared vocabulary lets scientists describe gene roles consistently.

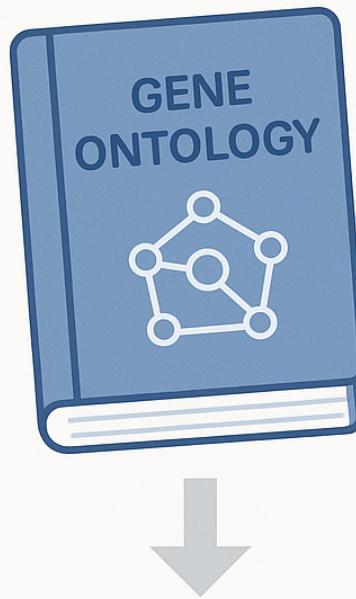
GO Enrichment

Gene Ontology (GO)

GO is like a dictionary of gene jobs

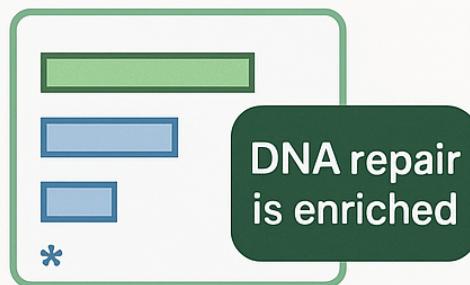
Biological Process
DNA repair
metabolism

Molecular Function
bind DNA
cut proteins



GO Enrichment

Are certain jobs showing up more often than expected by chance?



suggesting it's biologically important

When you have a big list of genes (e.g. from an experiment with significant gene ratios), GO enrichment looks for patterns:

“Are certain jobs (GO terms) showing up more often than expected by chance?”

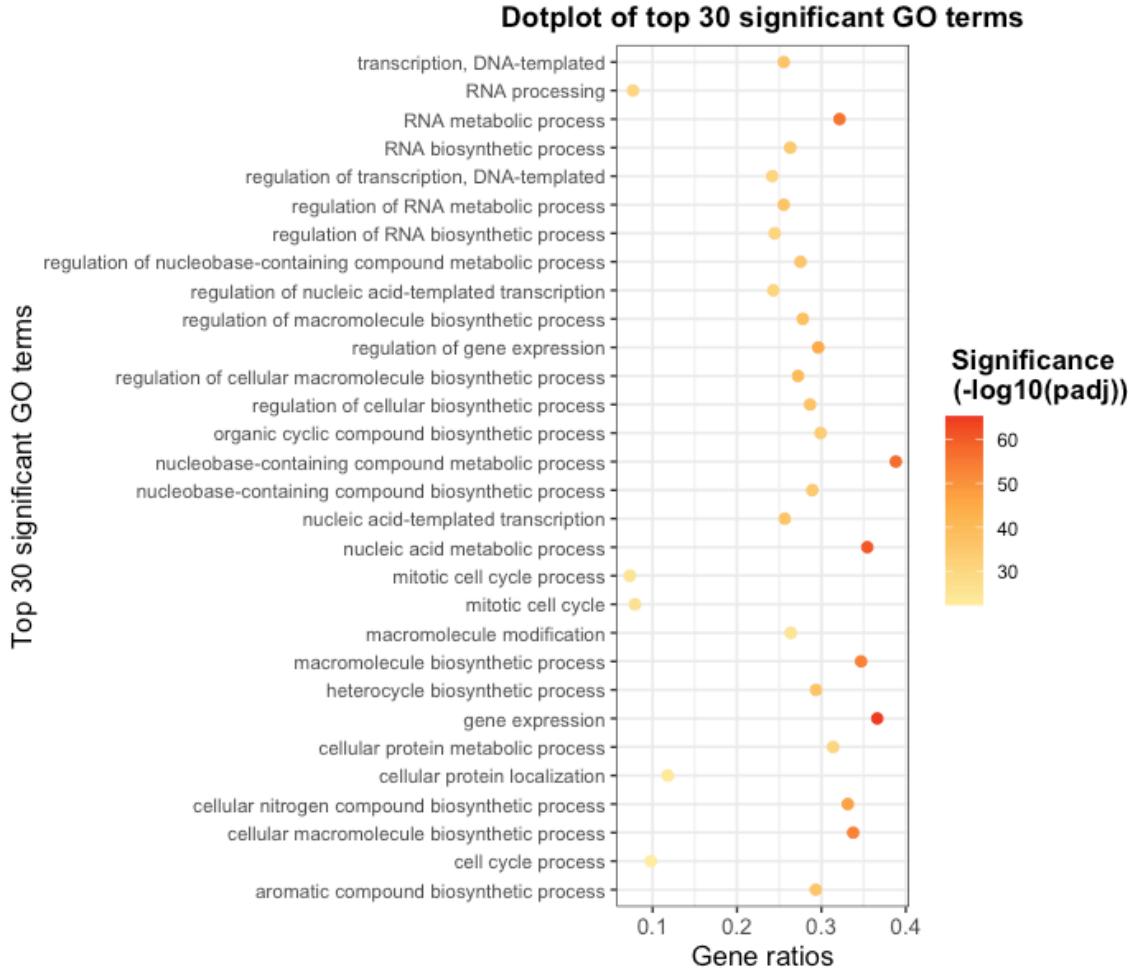
Example: If 20 out of 100 of your genes are about DNA repair, but normally only 5% of genes are DNA repair genes, then DNA repair is enriched → suggesting it's biologically important in your experiment.

In one sentence: GO tells us what jobs genes have, and GO enrichment finds out which jobs are over-represented in your gene list, revealing the bigger biological story.

![Alt text](img/mov10_FA.png{ width=1200px })

16.6 Analysis goal and workflow

Goal: Visually compare the most significant biological processes (BP) based on the number of associated differentially expressed genes (gene ratios) and significance values by creating the following plot:



The gene ratio here is just the fraction of genes in the significant list that are annotated to a specific GO term, so it doesn't say anything about the p-value.

To wrangle our data in preparation for the plotting, we are going to use the Tidyverse suite of tools to wrangle and visualize our data through several steps:

1. Read in the functional analysis results
2. Extract only the GO biological processes (BP) of interest
3. Select only the columns needed for visualization
4. Order by significance (p-adjusted values)
5. Rename columns to be more intuitive
6. Create additional metrics for plotting (e.g. gene ratios)

7. Plot results

Tidyverse tools

We are going to investigate more deeply the packages loaded with tidyverse, specifically the reading (`readr`), wrangling (`dplyr`), and plotting (`ggplot2`) tools.

16.6.1 1. Read in the functional analysis results

While the base R packages have perfectly fine methods for reading in data, the `readr` and `readxl` Tidyverse packages offer additional methods for reading in data. Let's read in our tab-delimited functional analysis results using `read_delim()`:

```
# Read in the functional analysis results
functional_GO_results <- read_delim(file = "data/gprofiler_results_Mov10oe.tsv",
  delim = "\t")

# Take a look at the results
functional_GO_results
```


16.7 # A tibble: 3,644 x 14

16.8 query.number significant p.value term.size query.size overlap.size recall

16.9

16.10 1 1 TRUE 0.00434 111 5850 52 0.009

16.11 2 1 TRUE 0.0033 110 5850 52 0.009

16.12 3 1 TRUE 0.0297 39 5850 21 0.004

16.13 4 1 TRUE 0.0193 70 5850 34 0.006

16.14 5 1 TRUE 0.0148 26 5850 16 0.003

16.15 6 1 TRUE 0.0187 22 5850 14 0.002

16.16 7 1 TRUE 0.0226 48 5850 25 0.004

16.17 8 1 TRUE 0.0491 17 5850 11 0.002

16.18 9 1 TRUE 0.00798 164 5850 71 0.012

16.19 10 1 TRUE 0.0439 19 5850 12 0.002

16.20 # i 3,634 more rows

16.21 # i 7 more variables: precision , term.id , domain ,

16.22 # subgraph.number , term.name , relative.depth ,

16.23 # intersection

16.23.1 How to do this in base R

```
# Read in the functional analysis results
functional_GO_results <- read.delim(file = "data/gprofiler_results_Mov10oe.tsv", sep = "\t")
# Take a look at the results
functional_GO_results
```

Notice that the results were automatically read in as a tibble and the output gives the number of rows, columns and the data type for each of the columns.

16.23.2 2. Extract only the GO biological processes (BP) of interest

Now that we have our data, we will need to wrangle it into a format ready for plotting.

To extract the biological processes of interest, we only want those rows where the `domain` is equal to `BP`, which we can do using the `filter()` function from the `dplyr` package.

To filter rows of a data frame/tibble based on values in different columns, we give a logical expression as input to the `filter()` function to return those rows for which the expression is TRUE.

Now let's return only those rows that have a `domain` of `BP`:

```
# Return only GO biological processes
bp_oe <- functional_GO_results %>%
  filter(domain == "BP")

# View(bp_oe)
```

16.23.3 How to do this in base R

```
{r} # Return only GO biological processes idx <- functional_GO_results$domain == "BP" bp_oe2 <- functional_GO_results[idx,
# View(bp_oe)
```

Now we have returned only those rows with a `domain` of `BP`. How have the dimensions of our results changed?**

Exercise:

We would like to perform an additional round of filtering to only keep the most specific GO terms.

1. For `bp_oe`, use the `filter()` function to only keep those rows where the `relative.depth` is greater than 4.
 2. Save output to overwrite our `bp_oe` variable.
-

16.23.4 3. Select only the columns needed for visualization

For visualization purposes, we are only interested in the columns related to the GO terms, the significance of the terms, and information about the number of genes associated with the terms.

To extract these columns from a data frame/tibble we can use the `select()` function. In contrast to base R, we do not need to put the column names in quotes for selection.

```
# Selecting columns to keep
bp_oe <- bp_oe %>%
  select(term.id, term.name, p.value, query.size, term.size, overlap.size, intersection)

head(bp_oe)
```

16.24 # A tibble: 6 x 7

16.25 term.id term.name p.value query.size term.size overlap.size intersection

16.26

16.27 1 GO:0032606 type I inte~ 0.00434 5850 111 52 rnf216,polr~

16.28 2 GO:0032479 regulation ~ 0.0033 5850 110 52 rnf216,polr~

16.29 3 GO:0032480 negative re~ 0.0297 5850 39 21 rnf216,otud~

16.30 4 GO:0032481 positive re~ 0.0193 5850 70 34 polr3b,polr~

16.31 5 GO:0010644 cell commun~ 0.0148 5850 26 16 pkp2,prkaca~

16.32 6 GO:0086064 cell commun~ 0.0187 5850 22 14 pkp2,prkaca~

16.32.1 How to do this in base R

```
# Selecting columns to keep
bp_oe <- bp_oe[, c("term.id", "term.name", "p.value", "query.size", "term.size", "overlap.size", "intersection")]
# View(bp_oe)
```

term.id	term.name	p.value	query.size	term.size	overl...
GO:0090672	telomerase RNA localization	2.41e-02	5850		16
GO:0090670	RNA localization to Cajal body	2.41e-02	5850		16
GO:0090671	telomerase RNA localization to Cajal body	2.41e-02	5850		16
GO:0071702	organic substance transport	7.90e-11	5850		2629
GO:0015931	nucleobase-containing compound transport	4.43e-05	5850		200
GO:0050657	nucleic acid transport	3.67e-06	5850		166
GO:0050658	RNA transport	3.67e-06	5850		166
GO:0051031	tRNA transport	4.88e-02	5850		33
GO:0051028	mRNA transport	2.48e-05	5850		137
GO:0016192	vesicle-mediated transport	1.39e-04	5850		1492
GO:0098876	vesicle-mediated transport to the plasma membrane	2.72e-03	5850		89
GO:0048193	Golgi vesicle transport	4.49e-05	5850		336
GO:0006890	retrograde vesicle-mediated transport, Golgi to ER	9.50e-05	5850		80
GO:0006892	post-Golgi vesicle-mediated transport	1.64e-02	5850		90
GO:0007034	vacuolar transport	3.30e-03	5850		110

16.32.2 4. Order GO processes by significance (adjusted p-values)

Now that we have only the rows and columns of interest, let's arrange these by significance, which is denoted by the adjusted p-value.

Let's sort the rows by adjusted p-value with the `arrange()` function from the `dplyr` package.

```
# Order by adjusted p-value ascending
bp_oe <- bp_oe %>%
  arrange(p.value)
```

16.32.3 5. Rename columns to be more intuitive

While not necessary for our visualization, renaming columns more intuitively can help with our understanding of the data using the `rename()` function from the `dplyr` package. The syntax is `new_name = old_name`.

Let's rename the `term.id` and `term.name` columns.

```
# Provide better names for columns
bp_oe <- bp_oe %>%
  dplyr::rename(GO_id = term.id, GO_term = term.name)
```

16.32.4 How to do this in base R

```
# Provide better names for columns
colnames(bp_oe)[colnames(bp_oe) == "term.id"] <- "GO_id"
colnames(bp_oe)[colnames(bp_oe) == "term.name"] <- "GO_term"
```

NOTE: In the case of two packages with identical function names, you can use `::` with the package name before and the function name after (e.g `stats::filter()`) to ensure that the correct function is implemented. The `::` can also be used to bring in a function from a library without loading it first.

In the example above, we wanted to use the `rename()` function specifically from the `dplyr` package, and not any of the other packages (or base R) which may have the `rename()` function.

Exercise

Rename the `intersection` column to `genes` to reflect the fact that these are the DE genes associated with the GO process.

16.32.5 6. Create additional metrics for plotting (e.g. gene ratios)

Finally, before we plot our data, we need to create a couple of additional metrics. The `mutate()` function enables you to create a new column from an existing column.

Let's generate gene ratios to reflect the number of DE genes associated with each GO process relative to the total number of DE genes.

```
# Create gene ratio column based on other columns in dataset
bp_oe <- bp_oe %>%
  mutate(gene_ratio = overlap.size/query.size)
```

16.32.6 How to do this in base R

```
# Create gene ratio column based on other columns in dataset
bp_oe <- cbind(bp_oe, gene_ratio = bp_oe$overlap.size / bp_oe$query.size)
```

Exercise

Create a column in `bp_oe` called `term_percent` to determine the percent of DE genes associated with the GO term relative to the total number of genes associated with the GO term (`overlap.size / term.size`)

Our final data for plotting should look like the table below:

go_id	go_term	p.value	query.size	term.size	overlap.size	genes
GO:0010467	gene expression	6.71e-66	5850	5257	2142	gclc,nfyα,b
GO:0090304	nucleic acid metabolic process	1.18e-61	5850	5103	2073	gclc,nfyα,d
GO:0006139	nucleobase-containing compound metabolic process	2.49e-58	5850	5731	2271	dpm1,gclc
GO:0016070	RNA metabolic process	7.28e-57	5850	4597	1881	gclc,nfyα,c
GO:0009059	macromolecule biosynthetic process	3.12e-54	5850	5066	2030	dpm1,gclc
GO:0034645	cellular macromolecule biosynthetic process	5.60e-54	5850	4907	1975	dpm1,gclc
GO:0044271	cellular nitrogen compound biosynthetic process	2.10e-47	5850	4882	1938	gclc,nfyα,s
GO:0010468	regulation of gene expression	4.25e-46	5850	4297	1733	gclc,nfyα,d
GO:2000112	regulation of cellular macromolecule biosynthetic pro...	1.22e-40	5850	3960	1593	gclc,nfyα,d
GO:0010556	regulation of macromolecule biosynthetic process	2.22e-39	5850	4073	1626	gclc,nfyα,d
GO:0031326	regulation of cellular biosynthetic process	4.08e-37	5850	4251	1676	gclc,nfyα,p
GO:0018130	heterocycle biosynthetic process	4.30e-37	5850	4375	1718	gclc,nfyα,s
GO:0051252	regulation of RNA metabolic process	7.61e-37	5850	3723	1494	gclc,nfyα,d
GO:0097659	nucleic acid-templated transcription	1.01e-36	5850	3745	1501	gclc,nfyα,d
GO:0019438	aromatic compound biosynthetic process	1.76e-36	5850	4378	1716	gclc,nfyα,s

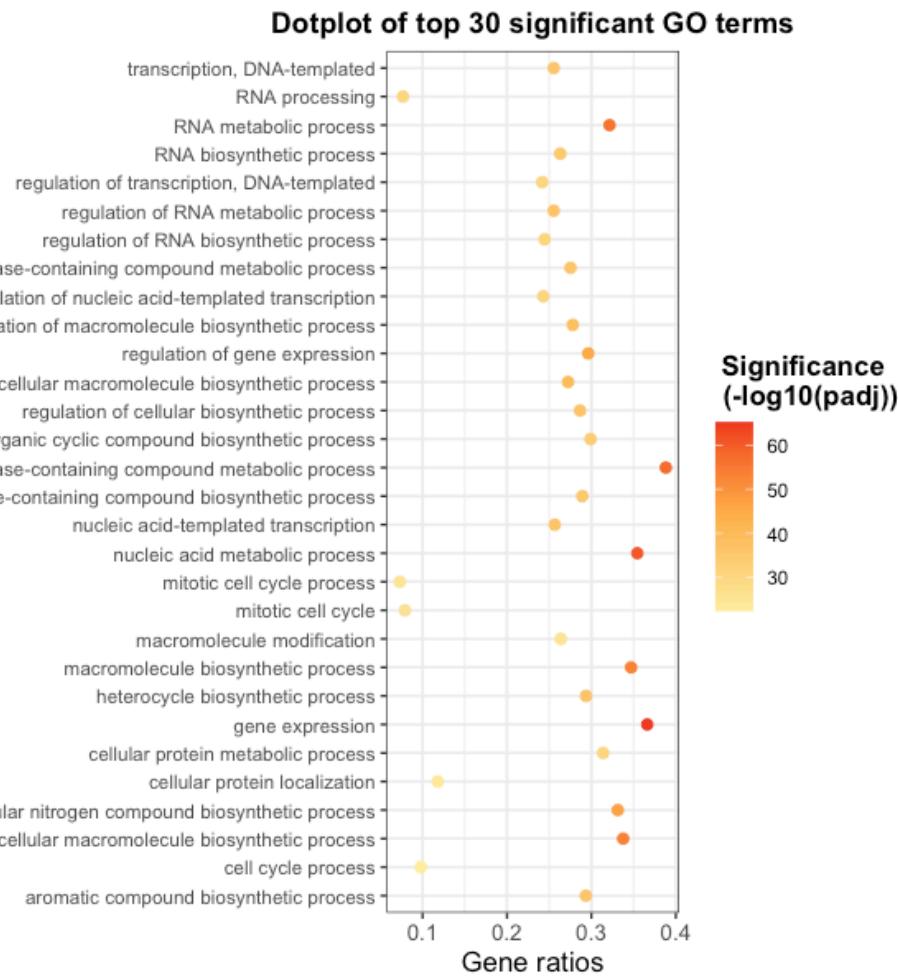
16.33 Next steps

Now that we have our results ready for plotting, we can use the `ggplot2` package to plot our results.

Homework

Use `ggplot2` to plot `bp_oe` to recapitulate the GO enrichment figure shown below using the top 30 categories:

Top 30 significant GO terms



16.33.1 Additional resources

- R for Data Science
 - teach the tidyverse
 - tidy style guide
-

Chapter 17

Codebook for IntroR and RStudio

69 points total

17.1 01 Setting up R and RStudio

Let's start by going to the environment panel in RStudio and clearing the environment. This is a good habit to get into when starting a new project, as it ensures that you are starting with a clean slate.

17.2 02 Introduction to R and RStudio

.md file = 02-introR-R-and-RStudio.md

17.2.1 Running commands and annotating code:

Add some additional code to the following chunk:

```
## Intro to R Lesson  
3 + 5
```

17.3 [1] 8

The chunk will run inline and output inline, but I prefer to set the chunk so it will run in the console, because it gives us extra room, especially for plotting. You can set this using the cog wheel at the top of the script window next to the Knit button.

17.3.1 The assignment operator (<-)

The assignment operator assigns values on the right to variables on the left.

In RStudio the keyboard shortcut for the assignment operator <- is Alt + - (Windows) or Option + - (Mac).

```
x <- 3
y <- 5
```

```
x + y
```

17.4 [1] 8

```
number <- x + y
number
```

17.5 [1] 8

Exercise points +2

1. Try changing the value of the variable x to 5. What happens to number?

```
# Your code here
```

- Ans:

2. Now try changing the value of variable y to contain the value 10. What do you need to do, to update the variable number?

```
# Your code here
```

- Ans:

17.6 03 R syntax and data structures

.md file = 03-introR-syntax-and-data-structures.md

17.6.1 Vectors

```
# Create a numeric vector and store the vector as a variable called 'glengths'
glengths <- c(4.6, 3000, 50000)
glengths
```

17.7 [1] 4.6 3000.0 50000.0

```
# Create a character vector and store the vector as a variable called 'species'
species <- c("ecoli", "human", "corn")
species
```

17.8 [1] "ecoli" "human" "corn"

Exercise points +1

Try to create a vector of numeric and character values by combining the two vectors that we just created (`glengths` and `species`). Assign this combined vector to a new variable called `combined`. Hint: you will need to use the combine `c()` function to do this. Print the combined vector in the console, what looks different compared to the original vectors?

```
# Your code here
```

- Ans:

17.8.1 Factors

Create a character vector and store the vector as a variable called ‘expression’

```
expression <- c("low", "high", "medium", "high", "low", "medium", "high")
expression
```

17.9 [1] "low" "high" "medium" "high" "low" "medium" "high"

Turn ‘expression’ vector into a factor

```
expression <- factor(expression)
expression
```

17.10 [1] low high medium high low medium high**17.11 Levels: high low medium**

```
length(expression)
```

17.12 [1] 7

```
levels(expression)
```

17.13 [1] "high" "low" "medium"

Exercises points +2

Let's say that in our experimental analyses, we are working with three different sets of cells: normal, cells knocked out for geneA (a very exciting gene), and cells overexpressing geneA. We have three replicates for each celltype.

1. Create a vector named samplegroup with nine elements: 3 control ("CTL") values, 3 knock-out ("KO") values, and 3 over-expressing ("OE") values.

```
# Your code here
```

2. Turn samplegroup into a factor data structure.

```
# Your code here
```

17.13.1 Dataframes

A `data.frame` is the de facto data structure for most tabular data and what we use for statistics and plotting. A `data.frame` is similar to a `matrix` in that it's a collection of vectors of the same length and each vector represents a column. However, in a `dataframe` each vector can be of a different data type (e.g., characters, integers, factors). A `dataframe` can be created using the `data.frame()` function.

```
# Create a data frame and store it as a variable called 'df'
df <- data.frame(species, glengths)
df
```

17.14 species glengths

17.15 1 ecoli 4.6

17.16 2 human 3000.0

17.17 3 corn 50000.0

Exercise points +1

Create a data frame called `favorite_books` with the following vectors as columns:

```
titles <- c("Catch-22", "Pride_and_Prejudice", "Nineteen_Eighty_Four")
pages <- c(453, 432, 328)
```

```
# Your code here
```

17.18 04 Functions in R

.md file = 04-introR-functions-and-arguments.md

17.18.1 Basic functions

We have already used a few examples of basic functions in the previous lessons i.e `c()`, and `factor()`.

Many of the base functions in R involve mathematical operations. One example would be the function `sqrt()`. The input/argument must be a number, and the output is the square root of that number. Let's try finding the square root of 81:

```
sqrt(81)
```

17.19 [1] 9

```
sqrt(glengths)
```

17.20 [1] 2.144761 54.772256 223.606798

Round function:

```
round(3.14159)
```

17.21 [1] 3

```
round(3.14159, digits = 2)
```

17.22 [1] 3.14

You can also round in the opposite direction, for example, to the nearest 100 by setting the digits argument to -2:

```
round(367.14159, digits = -2)
```

17.23 [1] 400

17.23.1 Seeking help on functions

```
?round
```

```
args(round)
```

17.24 function (x, digits = 0, ...)

17.25 NULL

```
example("round")
```


17.26

17.27 `round> round(.5 + -2:4) # IEEE / IEC rounding: -2 0 0 2 2 4 4`

17.28 `[1] -2 0 0 2 2 4 4`

17.29

17.30 `round> ## (this is good behaviour – do NOT report it as bug !)`

17.31 `round>`

17.32 `round> (x1 <- seq(-2, 4, by = .5))`

17.33 `[1] -2.0 -1.5 -1.0 -0.5 0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0`

17.34

17.35 `round> round(x1) #- IEEE / IEC rounding !`

17.36 `[1] -2 -2 -1 0 0 0 1 2 2 2 3 4 4`

17.37

17.38 `round> x1[trunc(x1) != floor(x1)]`

17.39 `[1] -1.5 -0.5`

17.40

17.41 `round> x1[round(x1) != floor(x1 + .5)]`

17.42 `[1] -1.5 0.5 2.5`

17.43

17.44 `round> (non.int <- ceiling(x1) != floor(x1))`

17.45 `[1] FALSE TRUE FALSE TRUE FALSE FALSE TRUE FALSE`

- Let's use base R function to calculate mean value of the `glengths` vector. You might need to search online to find what function can perform this task.

```
# Your code here
```

- Create a new vector `test <- c(1, NA, 2, 3, NA, 4)`. Use the same base R function from exercise 1 (with addition of proper argument), and calculate the mean value of the `test` vector. The output should be **2.5**.

NOTE: In R, missing values are represented by the symbol `NA` (not available).

It's a way to make sure that users know they have missing data, and make a conscious decision on how to deal with it.

There are ways to ignore `NA` during statistical calculation, or to remove `NA` from the vector.

```
test <- c(1, NA, 2, 3, NA, 4)
# Your code here
```

- Another commonly used base function is `sort()`. Use this function to sort the `glengths` vector in descending order.

```
# Your code here
```

17.54.1 User-defined Functions

Let's try creating a simple example function. This function will take in a numeric value as input, and return the squared value.

```
square_it <- function(x) {
  square <- x * x
  return(square)
}
```

Once you run the code, you should see a function named `square_it` in the Environment panel (located at the top right of Rstudio interface). Now, we can use this function as any other base R functions. We type out the name of the function, and inside the parentheses we provide a numeric value `x`:

```
square_it(5)
```

17.55 [1] 25

Pretty simple, right? In this case, we only had one line of code that was run, but in theory you could have many lines of code to get obtain the final results that you want to "return" to the user. We have only scratched the surface here when it comes to creating functions! If you are interested you can also find more detailed information on writing functions R-bloggers site.

Exercise points +2

1. Write a function called `multiply_it`, which takes two inputs: a numeric value `x`, and a numeric value `y`. The function will return the product of these two numeric values, which is `x * y`. For example, `multiply_it(x=4, y=6)` will return output 24.

Function:

```
# Your code here
```

Demo:

```
# Your code here
```

17.56 05 Packages and libraries

.md file = 05-introR_packages.md

17.56.1 Packages and Libraries

You can check what libraries (packages) are loaded in your current R session by typing into the console:

```
sessionInfo() #Print version information about R, the OS and attached or loaded packages
```


17.57 R version 4.5.1 (2025-06-13)

17.58 Platform: x86_64-apple-darwin20

17.59 Running under: macOS Sequoia 15.5

17.60

17.61 Matrix products: default

17.62 BLAS: /Library/Frameworks/R.framework/Versions/4.5-x86_64/Resources/lib/libRblas.0.dylib

17.63 LAPACK: /Library/Frameworks/R.framework/Versions/4.5-x86_64/Resources/lib/libRlapack.0.dylib
LAPACK version 3.12.1

17.64

17.65 locale:

17.66 [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

17.67

17.68 time zone: America/Los_Angeles

17.69 tzcode source: internal

17.70

17.71 attached base packages:

17.72 [1] stats graphics grDevices utils datasets methods base

17.73

17.74 other attached packages:

17.75 [1] lubridate_1.9.4 forcats_0.10.0 stringr_1.5.1 dplyr_1.1.4

```
# OR

search() #Gives a list of attached packages
```

17.91 [1] “.GlobalEnv” “package:lubridate” “package:forcats”

17.92 [4] “package:stringr” “package:dplyr” “package:purrr”

17.93 [7] “package:readr” “package:tidyverse” “package:tibble”

17.94 [10] “package:ggplot2” “package:tidyverse” “package:stats”

17.95 [13] “package:graphics” “package:grDevices” “package:utils”

17.96 [16] “package:datasets” “package:methods” “Autoloads”

17.97 [19] “package:base”

17.97.1 Installing packages

17.97.1.1 Packages from CRAN:

Previously we have introduced you to installing packages. An example is given below for the `ggplot2` package that will be required for some plots we will create later on. Run this code to install `ggplot2`.

Set eval = TRUE if you haven’t installed `ggplot2` yet

```
install.packages("ggplot2")
```

Alternatively, packages can also be installed from Bioconductor, a repository of packages which provides tools for the analysis and comprehension of high-throughput **genomic data**.

Set eval = TRUE if you haven’t installed `BiocManager` yet

```
install.packages("BiocManager")
```

17.97.1.2 Packages from Bioconductor:

Now you can use `BiocManager::install` to install packages available on Bioconductor. Bioconductor is a free, open source and open development software project for the analysis and comprehension of genomic data generated by wet lab experiments in molecular biology.

```
# DO NOT RUN THIS!
BiocManager::install("ggplot2")
```

17.97.2 Loading libraries/packages

```
library(ggplot2)
```

To see the functions in a package you can also type:

```
ls("package:ggplot2")
```


17.98 [1] "%+%" "%+replace%"

17.99 [3] "aes" "aes_"

17.100 [5] "aes_all" "aes_auto"

17.101 [7] "aes_q" "aes_string"

17.102 [9] "after_scale" "after_stat"

17.103 [11] "alpha" "annotate"

17.104 [13] "annotation_custom" "annotation_logticks"

17.105 [15] "annotation_map" "annotation_raster"

17.106 [17] "arrow" "as_label"

17.107 [19] "as_labeller" "autolayer"

17.108 [21] "autoplot" "AxisSecondary"

17.109 [23] "benchplot" "binned_scale"

17.110 [25] "borders" "calc_element"

17.111 [27] "check_device" "combine_vars"

17.112 [29] "continuous_scale" "Coord"

17.113 [31] "coord_cartesian" "coord_equal"

17.114 [33] "coord_fixed" "coord_flip"

17.115 [35] "coord_map" "coord_munch"

17.116 [37] "coord_polar" "coord_quickmap"

17.117 [39] "coord_sf" "coord_sf"

The tidyverse suite of integrated packages was designed to work together to make common data science operations more user-friendly. We will be using the tidyverse suite in later lessons, and so let's install it. Use the function from the `BiocManager` package.

Set `eval=TRUE` if tidyverse is not yet installed. How can you run the `BiocManager::install()` command for a package that is already installed without getting an error?

```
# Your code here
```

- Ans:

17.391 06 Subsetting: vectors and factors

.md file = 06-introR-data wrangling.md

17.391.1 Vectors: Selecting using indices

```
age <- c(15, 22, 45, 52, 73, 81)
age
```

17.392 [1] 15 22 45 52 73 81

```
age[5]
```

17.393 [1] 73

Reverse selection:

```
age[-5]
```

17.394 [1] 15 22 45 52 81

```
age[c(3, 5, 6)] ## nested
```

17.395 [1] 45 73 81

```
# OR
```

```
## create a vector first then select
idx <- c(3, 5, 6) # create vector of the elements of interest
age[idx]
```

17.396 [1] 45 73 81

```
age[1:4]
```

17.397 [1] 15 22 45 52

Exercises points +3, +4 for bonus

1. Create a vector called alphabets with the following letters, C, D, X, L, F.

```
alphabets <- c("C", "D", "X", "L", "F")
```

2. Use the associated indices along with [] to do the following:

- a) only display C, D and F

```
# Your code here
```

```
b) display all except X
```

```
# Your code here
```

```
c) display the letters in the opposite order (F, L, X, D, C) *Bonus points for using a function instead of the indices (Hint: use Google)
```

```
# Your code here
```

17.397.1 Selecting using logical operators

```
age
```

17.398 [1] 15 22 45 52 73 81

```
age > 50
```

17.399 [1] FALSE FALSE FALSE TRUE TRUE TRUE

```
age[age > 50] ## nested
```

17.400 [1] 52 73 81

```
age > 50 | age < 18 ## returns logical values for each element
```

17.401 [1] TRUE FALSE FALSE TRUE TRUE TRUE

```
age
```

17.402 [1] 15 22 45 52 73 81

```
age[age > 50 | age < 18] ## nested
```

17.403 [1] 15 52 73 81

```
# OR
## create a vector first then select
idx <- age > 50 | age < 18
idx
```

17.404 [1] TRUE FALSE FALSE TRUE TRUE TRUE

```
age[idx]
```

17.405 [1] 15 52 73 81

17.405.1 Logical operators with `which()`

```
which(age > 50 | age < 18) ## returns only the indices of the TRUE values
```

17.406 [1] 1 4 5 6

```
age[which(age > 50 | age < 18)] ## nested
```

17.407 [1] 15 52 73 81

139

17.407 [1] 15 52 73 81

```
# OR
```

```
## create a vector first then select
idx_num <- which(age > 50 | age < 18)
age[idx_num]
```

17.408 [1] 15 52 73 81

17.408.1 Factors

```
expression <- c("low", "high", "medium", "high", "low", "medium", "high")
expression <- factor(expression)
expression
```

17.409 [1] low high medium high low medium high

17.410 Levels: high low medium

```
str(expression)
```

17.411 Factor w/ 3 levels “high”,“low”,“medium”: 2 1 3 1 2 3 1

```
expression[expression == "high"] ## This will only return those elements in the factor equal to 'high'
```

17.412 [1] high high high

17.413 Levels: high low medium

```
exp <- expression[expression == "high"]
exp
```

17.414 [1] high high high

17.415 Levels: high low medium

Notice here that `exp` is a factor with only one level, “high”, yet `Levels` still shows all three levels. This is because the factor is a categorical variable, and the levels are the categories. In this case, the only category is “high”. If you want to remove unused levels, you can use `droplevels()`, or since there is only one level, you can use `as.character()` to convert it to a character vector.

```
droplevels(exp) # this will remove the unused levels
```

17.416 [1] high high high

17.417 Levels: high

```
as.character(exp) # this will convert the factor to a character vector
```

17.418 [1] “high” “high” “high”

Exercise points +1

Extract only those elements in `samplegroup` that are not KO (nesting the logical operation is optional).

```
samplegroup <- c(rep("CTL", 3), rep("KO", 3), rep("OE", 3))
samplegroup <- factor(samplegroup)
# Your code here
```

17.418.0.1 Releveling factors

```
expression # the default of the factor function is to order the levels alphabetically
```

17.419 [1] low high medium high low medium high

17.420 Levels: high low medium

```
expression <- factor(expression, levels = c("low", "medium", "high")) # you can re-factor a factor
str(expression)
```

17.421 Factor w/ 3 levels “low”,“medium”,...: 1 3 2 3 1 2 3

Exercise points +1

Use the samplegroup factor we created in a previous lesson, and relevel it such that KO is the first level followed by CTL and OE.

```
samplegroup <- c(rep("CTL", 3), rep("KO", 3), rep("OE", 3))
samplegroup <- factor(samplegroup)
# Your code here
```

17.422 07 Reading and data inspection

.md file = 07-reading_and_data_inspection.md

17.422.1 Reading data into R:

```
~?~(read.csv)
```

```
### Reading data into R
metadata <- read.csv(file = "data/mouse_exp_design.csv")
```

We can see if it has successfully been read in by running:

```
metadata
```

17.423 genotype celltype replicate

17.424 sample1 Wt typeA 1

17.425 sample2 Wt typeA 2

17.426 sample3 Wt typeA 3

17.427 sample4 KO typeA 1

17.428 sample5 KO typeA 2

17.429 sample6 KO typeA 3

17.430 sample7 Wt typeB 1

17.431 sample8 Wt typeB 2

17.432 sample9 Wt typeB 3

17.433 sample10 KO typeB 1

17.434 sample11 KO typeB 2

17.435 sample12 KO typeB 3

Exercise 1 points +2

1. Read “project-summary.txt” in to R using `read.table()` with the appropriate arguments and store it as the variable `proj_summary`. To figure out the appropriate arguments to use with `read.table()`, keep the following in mind:

- all the columns in the input text file have column name/headers
- you want the first column of the text file to be used as row names
- hint: look up the input for the `row.names = argument` in `read.table()`

```
# Your code here
```

2. Display the contents of proj_summary in your console

```
# Your code here
```

17.435.1 Inspecting data structures:

Getting info on environmental variables:

Let's use the metadata file that we created to test out data inspection functions.

```
head(metadata) # shows the first 6 rows of the data frame
```

17.436 genotype celltype replicate

17.437 sample1 Wt typeA 1

17.438 sample2 Wt typeA 2

17.439 sample3 Wt typeA 3

17.440 sample4 KO typeA 1

17.441 sample5 KO typeA 2

17.442 sample6 KO typeA 3

```
str(metadata) # shows the structure of the data frame
```

17.443 ‘data.frame’: 12 obs. of 3 variables:

17.444 \$ genotype : chr “Wt” “Wt” “Wt” “KO” ...

17.445 \$ celltype : chr “typeA” “typeA” “typeA” “typeA” ...

17.446 \$ replicate: int 1 2 3 1 2 3 1 2 3 1 ...

```
dim(metadata)
```

17.447 [1] 12 3

```
nrow(metadata) # shows the number of rows in the data frame
```

17.448 [1] 12

```
ncol(metadata)
```

17.449 [1] 3

```
class(metadata)
```

17.450 [1] “data.frame”

```
names(metadata)
```

17.451 [1] “genotype” “celltype” “replicate”

Exercise 2 points +2

- What is the class of each column in metadata (use one command)?

```
# Your code here
```

- What is the median of the replicates in metadata (use one command)?

```
# Your code here
```

Exercise 3 points +6

Use the `class()` function on `glengths` and `metadata`, how does the output differ between the two?

```
# Your code here
```

- Ans:

Use the `summary()` function on the `proj_summary` dataframe, what is the median “Intronic_Rate”?

```
# Your code here
```

- Ans:

How long is levels of the `samplegroup` factor?

```
# Your code here
```

- Ans:

What are the dimensions of the `proj_summary` dataframe?

```
# Your code here
```

- Ans:

When you use the `rownames()` function on metadata, what is the data structure of the output?

```
# Your code here
```

- Ans:

How many elements in (how long is) the output of `colnames(proj_summary)`? Don't count, but use another function to determine this.

```
# Your code here
```

- Ans:

17.452 08 Data frames and matrices

.md file = 08-introR-_data_wrangling2.md

17.452.1 Data wrangling: dataframes

Dataframes (and matrices) have 2 dimensions (rows and columns), so if we want to select some specific data from it we need to specify the “coordinates” we want from it. We use the same square bracket notation but rather than providing a single index, there are two indices required. Within the square bracket, row numbers come first followed by column numbers (and the two are separated by a comma).

Extracting values from data.frames:

```
# Extract value 'Wt'  
metadata[1, 1]
```

17.453 [1] “Wt”

```
# Extract third row  
metadata[3, ]
```

17.454 genotype celltype replicate

17.455 sample3 Wt typeA 3

```
# Extract third column  
metadata[, 3]
```

17.456 [1] 1 2 3 1 2 3 1 2 3 1 2 3

```
# Extract third column as a data frame  
metadata[, 3, drop = FALSE]
```

17.457 replicate

17.458 sample1 1

17.459 sample2 2

17.460 sample3 3

17.461 sample4 1

17.462 sample5 2

17.463 sample6 3

17.464 sample7 1

17.465 sample8 2

17.466 sample9 3

17.467 sample10 1

17.468 sample11 2

17.469 sample12 3

```
# Dataframe containing first two columns  
metadata[, 1:2]
```

17.470 genotype celltype

17.471 sample1 Wt typeA

17.472 sample2 Wt typeA

17.473 sample3 Wt typeA

17.474 sample4 KO typeA

17.475 sample5 KO typeA

17.476 sample6 KO typeA

17.477 sample7 Wt typeB

17.478 sample8 Wt typeB

17.479 sample9 Wt typeB

17.480 sample10 KO typeB

17.481 sample11 KO typeB

17.482 sample12 KO typeB

```
# Data frame containing first, third and sixth rows  
metadata[c(1, 3, 6), ]
```

17.483 genotype celltype replicate

17.484 sample1 Wt typeA 1

17.485 sample3 Wt typeA 3

17.486 sample6 KO typeA 3

```
# Extract the celltype column for the first three samples
metadata[c("sample1", "sample2", "sample3"), "celltype"]
```

17.487 [1] “typeA” “typeA” “typeA”

```
# Check column names of metadata data frame
colnames(metadata)
```

17.488 [1] “genotype” “celltype” “replicate”

```
# names and colnames are the same for data frames
# Check row names of metadata data frame
rownames(metadata)
```

17.489 [1] “sample1” “sample2” “sample3” “sample4” “sample5” “sample6”

17.490 [7] “sample7” “sample8” “sample9” “sample10” “sample11” “sample12”

```
# Extract the genotype column
metadata$genotype
```

17.491 [1] “Wt” “Wt” “Wt” “KO” “KO” “KO” “Wt” “Wt” “Wt” “KO” “KO” “KO”

```
# Extract the first five values/elements of the genotype column using the $ operator
metadata$genotype[1:5]
```

17.492 [1] “Wt” “Wt” “Wt” “KO” “KO”

Exercises points +3

Return a data frame with only the genotype and replicate column values for sample2 and sample8.

```
# Your code here
```

Return the fourth and ninth values of the replicate column.

```
# Your code here
```

Extract the replicate column as a data frame.

```
# Your code here
```

17.492.0.1 Selecting using logical operators:

For example, if we want to return only those rows of the data frame with the celltype column having a value of typeA, we would perform two steps:

1. Identify which rows in the celltype column have a value of typeA.
2. Use those TRUE values to extract those rows from the data frame.

To do this we would extract the column of interest as a vector, with the first value corresponding to the first row, the second value corresponding to the second row, so on and so forth. We use that vector in the logical expression. Here we are looking for values to be equal to typeA,

So our logical expression would be:

```
names(metadata)
```

17.493 [1] “genotype” “celltype” “replicate”

```
metadata$celltype
```

17.494 [1] “typeA” “typeA” “typeA” “typeA” “typeA” “typeA” “typeB” “typeB” “typeB”**17.495 [10] “typeB” “typeB” “typeB”**

```
table(metadata$celltype)
```

17.496**17.497 typeA typeB****17.498 6 6**

```
metadata$celltype == "typeA"
```

17.499 [1] TRUE TRUE TRUE TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE

```
logical_idx <- metadata$celltype == "typeA"
```

If we want to keep all the rows that have celltype == “typeA”: Any row in the logical_idx that is TRUE will be kept, those that are FALSE will be discarded.

```
metadata[logical_idx, ]
```

17.500 genotype celltype replicate**17.501 sample1 Wt typeA 1****17.502 sample2 Wt typeA 2****17.503 sample3 Wt typeA 3****17.504 sample4 KO typeA 1****17.505 sample5 KO typeA 2****17.506 sample6 KO typeA 3**

```
idx <- which(metadata$celltype == "typeA")
```

```
metadata[idx, ]
```

17.507 genotype celltype replicate

17.508 sample1 Wt typeA 1

17.509 sample2 Wt typeA 2

17.510 sample3 Wt typeA 3

17.511 sample4 KO typeA 1

17.512 sample5 KO typeA 2

17.513 sample6 KO typeA 3

```
# OR you can use nesting
metadata[which(metadata$celltype == "typeA"), ]
```

17.514 genotype celltype replicate

17.515 sample1 Wt typeA 1

17.516 sample2 Wt typeA 2

17.517 sample3 Wt typeA 3

17.518 sample4 KO typeA 1

17.519 sample5 KO typeA 2

17.520 sample6 KO typeA 3

Let's try another subsetting. Extract the rows of the metadata data frame for only the replicates 2 and 3. First, let's create the logical expression for the column of interest (replicate):

```
table(metadata$replicate)
```

17.521**17.522 1 2 3****17.523 4 4 4**

```
idx <- which(metadata$replicate > 1)
idx
```

17.524 [1] 2 3 5 6 8 9 11 12

```
metadata[idx, ]
```

17.525 genotype celltype replicate**17.526 sample2 Wt typeA 2****17.527 sample3 Wt typeA 3****17.528 sample5 KO typeA 2****17.529 sample6 KO typeA 3****17.530 sample8 Wt typeB 2****17.531 sample9 Wt typeB 3****17.532 sample11 KO typeB 2****17.533 sample12 KO typeB 3**

This should return the indices for the rows in the replicate column within metadata that have a value of 2 or 3. Now, we can save those indices to a variable and use that variable to extract those corresponding rows from the metadata table. Alternatively, you could do this in one line:

```
sub_meta <- metadata[which(metadata$replicate > 1), ]
```

Exercise points +1

Subset the metadata dataframe to return only the rows of data with a genotype of KO.

```
# Your code here
```

17.534 09 Logical operators for matching

.md file = 09-identifying-matching-elements.md

17.534.1 Logical operators to match elements

```
rpkm_data <- read.csv("data/counts.rpkm.csv")
```

```
head(rpkm_data) # this is our counts data
```


17.535 sample2 sample5 sample7 sample8 sample9 sample4

17.536 ENSMUSG00000000001 19.265000 23.7222000 2.611610 5.8495400
6.5126300 24.076700

17.537 ENSMUSG00000000003 0.000000 0.0000000 0.000000 0.0000000
0.0000000 0.000000

17.538 ENSMUSG00000000028 1.032290 0.8269540 1.134410 0.6987540
0.9251170 0.827891

17.539 ENSMUSG00000000031 0.000000 0.0000000 0.000000 0.0298449
0.0597726 0.000000

17.540 ENSMUSG00000000037 0.056033 0.0473238 0.000000 0.0685938
0.0494147 0.180883

17.541 ENSMUSG00000000049 0.258134 1.0730200 0.252342 0.2970320
0.2082800 2.191720

17.542 sample6 sample12 sample3 sample11 sample10

17.543 ENSMUSG00000000001 20.8198000 26.9158000 20.889500 24.0465000
24.198100

17.544 ENSMUSG00000000003 0.0000000 0.0000000 0.000000 0.0000000
0.000000

17.545 ENSMUSG00000000028 1.1686300 0.6735630 0.892183 0.9753270
1.045920

17.546 ENSMUSG00000000031 0.0511932 0.0204382 0.000000 0.0000000
0.000000

17.547 ENSMUSG00000000037 0.1438840 0.0662324 0.146196 0.0206405
0.017004

17.548 ENSMUSG00000000049 1.6853800 0.1161970 0.421286 0.0634322
0.369550

```
colnames(rpkm_data)
```

17.556 [1] "sample2" "sample5" "sample7" "sample8" "sample9" "sample4"

17.557 [7] "sample6" "sample12" "sample3" "sample11" "sample10" "sample1"

```
rownames(metadata) # this is our metadata describing the count data samples
```

17.558 [1] "sample1" "sample2" "sample3" "sample4" "sample5" "sample6"

17.559 [7] "sample7" "sample8" "sample9" "sample10" "sample11" "sample12"

It looks as if the sample names (header) in our data matrix are similar to the row names of our metadata file, but it's hard to tell since they are not in the same order.

```
ncol(rpkm_data)
```

17.560 [1] 12

```
nrow(metadata)
```

17.561 [1] 12

17.561.1 The %in% operator

The `%in%` operator will take each element from vector1 as input, one at a time, and evaluate if the element is present in vector2. The two vectors do not have to be the same size. This operation will return a vector containing logical values to indicate whether or not there is a match. The new vector will be of the same length as vector1. Take a look at the example below:

```
A <- c(1, 3, 5, 7, 9, 11) # odd numbers
B <- c(2, 4, 6, 8, 10, 12) # even numbers

# test to see if each of the elements of A is in B
A %in% B
```

17.562 [1] FALSE FALSE FALSE FALSE FALSE FALSE

Let's change a couple of numbers inside vector B to match vector A:

```
A <- c(1, 3, 5, 7, 9, 11) # odd numbers
B <- c(2, 4, 6, 8, 1, 5) # add some odd numbers in

# test to see if each of the elements of A is in B
A %in% B
```

17.563 [1] TRUE FALSE TRUE FALSE FALSE FALSE

```
intersection <- A %in% B
intersection
```

17.564 [1] TRUE FALSE TRUE FALSE FALSE FALSE

```
A[intersection]
```

17.565 [1] 1 5

```
any(A %in% B)
```

17.566 [1] TRUE

```
all(A %in% B)
```

17.567 [1] FALSE

Exercise points +2

1. Using the A and B vectors created above, evaluate each element in B to see if there is a match in A

```
# Your code here
```

2. Subset the B vector to only return those values that are also in A.

```
# Your code here
```

17.568 [1] TRUE TRUE TRUE TRUE TRUE

159

Suppose we had two vectors containing same values. **How can we check if those values are in the same order in each vector?** In this case, we can use == operator to compare each element of the same position from two vectors. Unlike %in% operator, == operator requires that two vectors are of equal length.

```
A <- c(10, 20, 30, 40, 50)
B <- c(50, 40, 30, 20, 10) # same numbers but backwards

# test to see if each element of A is in B
A %in% B
```

17.568 [1] TRUE TRUE TRUE TRUE TRUE

```
# test to see if each element of A is in the same position in B
A == B
```

17.569 [1] FALSE FALSE TRUE FALSE FALSE

```
# use all() to check if they are a perfect match
all(A == B)
```

17.570 [1] FALSE

Let's try this on our genomic data, and see whether we have metadata information for all samples in our expression data.

```
x <- rownames(metadata)
y <- colnames(rpkm_data)
```

```
all(x %in% y)
```

17.571 [1] TRUE

Now we can replace x and y by their real values to get the same answer:

```
all(rownames(metadata) %in% colnames(rpkm_data))
```

17.572 [1] TRUE

```
x == y
```

17.573 [1] FALSE FALSE

```
all(x == y)
```

17.574 [1] FALSE

Looks like all of the samples are there, but they need to be reordered. To reorder our genomic samples, we will learn different ways to reorder data in our next lesson.

Exercise points +3

We have a list of 6 marker genes that we are very interested in. Our goal is to extract count data for these genes using the %in% operator from the rpkm_data data frame, instead of scrolling through rpkm_data and finding them manually.

First, let's create a vector called important_genes with the Ensembl IDs of the 6 genes we are interested in:

1. Use the %in% operator to determine if all of these genes are present in the row names of the rpkm_data data frame.

```
# Your code here
```

2. Extract the rows from rpkm_data that correspond to these 6 genes using [] and the %in% operator. Double check the row names to ensure that you are extracting the correct rows.

```
# Your code here
```

3. Extract the rows from rpkm_data that correspond to these 6 genes using [], but without using the %in% operator.

```
# Your code here
```

17.575 10 Reordering to match datasets

.md file = 10-reordering-to-match-datasets.md

Exercise points +1

We know how to reorder using indices, let's try to use it to reorder the contents of one vector to match the contents of another. Let's create the vectors first and second as detailed below:

```
first <- c("A", "B", "C", "D", "E")
second <- c("B", "D", "E", "A", "C") # same letters but different order
```

```
first
```

17.576 [1] "A" "B" "C" "D" "E"

161

17.576 [1] "A" "B" "C" "D" "E"

```
second
```

17.577 [1] "B" "D" "E" "A" "C"

How would you reorder the second vector to match first?

```
# Your code here
```

17.577.1 The `match` function

`match()` takes 2 arguments. The first argument is a vector of values in the order you want, while the second argument is the vector of values to be reordered such that it will match the first:

1st vector: values in the order you want 2nd vector: values to be reordered

The `match` function returns the position of the matches (indices) with respect to the second vector, which can be used to re-order it so that it matches the order in the first vector. Let's use `match()` on the first and second vectors we created.

```
# Saving indices for how to reorder `second` to match `first`
reorder_idx <- match(first, second)
reorder_idx
```

17.578 [1] 4 1 5 2 3

```
# Reordering the second vector to match the order of the first vector
second[reorder_idx]
```

17.579 [1] "A" "B" "C" "D" "E"

```
# Reordering and saving the output to a variable
second_reordered <- second[reorder_idx]
second_reordered == first
```

17.580 [1] TRUE TRUE TRUE TRUE TRUE

Matching with vectors of different lengths:

```
first <- c("A", "B", "C", "D", "E")
second <- c("D", "B", "A") # remove values
```

```
match(first, second)
```

17.581 [1] 3 2 NA 1 NA

```
second[match(first, second)]
```

17.582 [1] “A” “B” NA “D” NA

NOTE: For values that don't match by default return an NA value. You can specify what values you would have it assigned using nomatch argument.

Example:

```
mtch <- match(first, second, nomatch = 0)
mtch
```

17.583 [1] 3 2 0 1 0

```
second[mtch]
```

17.584 [1] “A” “B” “D”

```
# Check row names of the metadata
rownames(metadata)
```

17.585 [1] “sample1” “sample2” “sample3” “sample4” “sample5” “sample6”

17.586 [7] “sample7” “sample8” “sample9” “sample10” “sample11” “sample12”

```
# Check the column names of the counts data
colnames(rpkm_data)
```

17.587 [1] “sample2” “sample5” “sample7” “sample8” “sample9” “sample4”

17.588 [7] “sample6” “sample12” “sample3” “sample11” “sample10” “sample1”

Reordering genomic data using the `match()` function:

17.589 [1] 12 1 9 6 2 7 3 4 5 11 10 8

163

```
# Use the match function to reorder the counts data frame to have the sample  
# names in the same order as the metadata data frame, so rownames(metadata)  
# comes first, order is important  
genomic_idx <- match(rownames(metadata), colnames(rpkm_data))  
  
genomic_idx
```

17.589 [1] 12 1 9 6 2 7 3 4 5 11 10 8

```
# Reorder the counts data frame to have the sample names in the same order as  
# the metadata data frame  
rpkm_ordered <- rpkm_data[, genomic_idx]
```

```
# View the reordered counts View(rpkm_ordered)  
  
# I prefer to use `head` -- and set the output to a limited number of columns  
# so that it fits in the console  
  
head(rpkm_ordered[, 1:5])
```

17.590 sample1 sample2 sample3 sample4 sample5

17.591 ENSMUSG00000000001 19.7848000 19.265000 20.889500 24.076700
23.7222000

17.592 ENSMUSG00000000003 0.0000000 0.0000000 0.0000000 0.0000000
0.0000000

17.593 ENSMUSG00000000028 0.9377920 1.032290 0.892183 0.827891
0.8269540

17.594 ENSMUSG00000000031 0.0359631 0.0000000 0.0000000 0.0000000
0.0000000

17.595 ENSMUSG00000000037 0.1514170 0.056033 0.146196 0.180883
0.0473238

17.596 ENSMUSG00000000049 0.2567330 0.258134 0.421286 2.191720
1.0730200

Now we can check if the rownames of metadata is in the same order as the colnames of rpkm_ordered:

```
all(rownames(metadata) == colnames(rpkm_ordered))
```

17.597 [1] TRUE

```
# important to check that the order is correct!!!
```

Exercises points +2

After talking with your collaborator, it becomes clear that sample2 and sample9 were actually from a different mouse background than the other samples and should not be part of our analysis. Create a new variable called subset_rpkm that has these columns removed from the rpkm_ordered data frame.

```
# Your code here
```

Use the `match()` function to subset the metadata data frame so that the row names of the metadata data frame match the column names of the subset_rpkm data frame.

```
# Your code here
```

17.598 11 Plotting and data visualization

.md file = 11-setting_up_to_plot.md10-reordering-to-match-datasets.md

17.598.1 Setup a data frame for visualization

```
mean(rpkm_ordered$sample1)
```

17.599 [1] 10.2661

But what we want is a vector of 12 values that we can add to the metadata data frame. What is the best way to do this?

To get the mean of all the samples in a single line of code the `map()` family of function is a good option.

17.599.1 The `map` family of functions

We can use map functions to execute some task/function on every element in a vector, or every column in a dataframe, or every component of a `list`, and so on.

- `map()` creates a list.
- `map_lgl()` creates a logical vector.
- `map_int()` creates an integer vector.
- `map_dbl()` creates a “double” or numeric vector.
- `map_chr()` creates a character vector.

The syntax for the `map()` family of functions is:

- `map(object, function_to_apply)`

We can use the `map_dbl()` to get the mean of each column of `rpkm_ordered` in just one command:

```
# install purrr if you need to by uncommenting the following line:
# BiocManager::install(purrr)
library(purrr) # Load the purrr

samplemeans <- map_dbl(rpkm_ordered, mean)
```

The output of `map_dbl()` is a *named* vector of length 12.

17.599.2 Adding to a metadata object

Before we add `samplemeans` as a new column to metadata, let’s create a vector with the ages of each of the mice in our data set.

```
# Create a numeric vector with ages. Note that there are 12 elements here
age_in_days <- c(40, 32, 38, 35, 41, 32, 34, 26, 28, 28, 30, 32)
```

Now, we are ready to combine the `metadata` data frame with the 2 new vectors to create a new data frame with 5 columns:

```
# Add the new vectors as the last columns to the metadata
new_metadata <- data.frame(metadata, samplemeans, age_in_days)

# Take a look at the new_metadata object
head(new_metadata)
```

17.600 genotype celltype replicate samplemeans age_in_days

17.601 sample1 Wt typeA 1 10.266102 40

17.602 sample2 Wt typeA 2 10.849759 32

17.603 sample3 Wt typeA 3 9.452517 38

17.604 sample4 KO typeA 1 15.833872 35

17.605 sample5 KO typeA 2 15.590184 41

17.606 sample6 KO typeA 3 15.551529 32

17.607 12 Data visualization with ggplot2

.md file = 12-ggplot2.md

```
## load the new_metadata data frame into your environment from a .RData object,
## if you need to set eval = TRUE
load("data/new_metadata.RData")
```

```
# this data frame should have 12 rows and 5 columns
head(new_metadata)
```

17.608 genotype celltype replicate samplemeans age_in_days

17.609 sample1 Wt typeA 1 10.266102 40

17.610 sample2 Wt typeA 2 10.849759 32

17.611 sample3 Wt typeA 3 9.452517 38

17.612 sample4 KO typeA 1 15.833872 35

17.613 sample5 KO typeA 2 15.590184 41

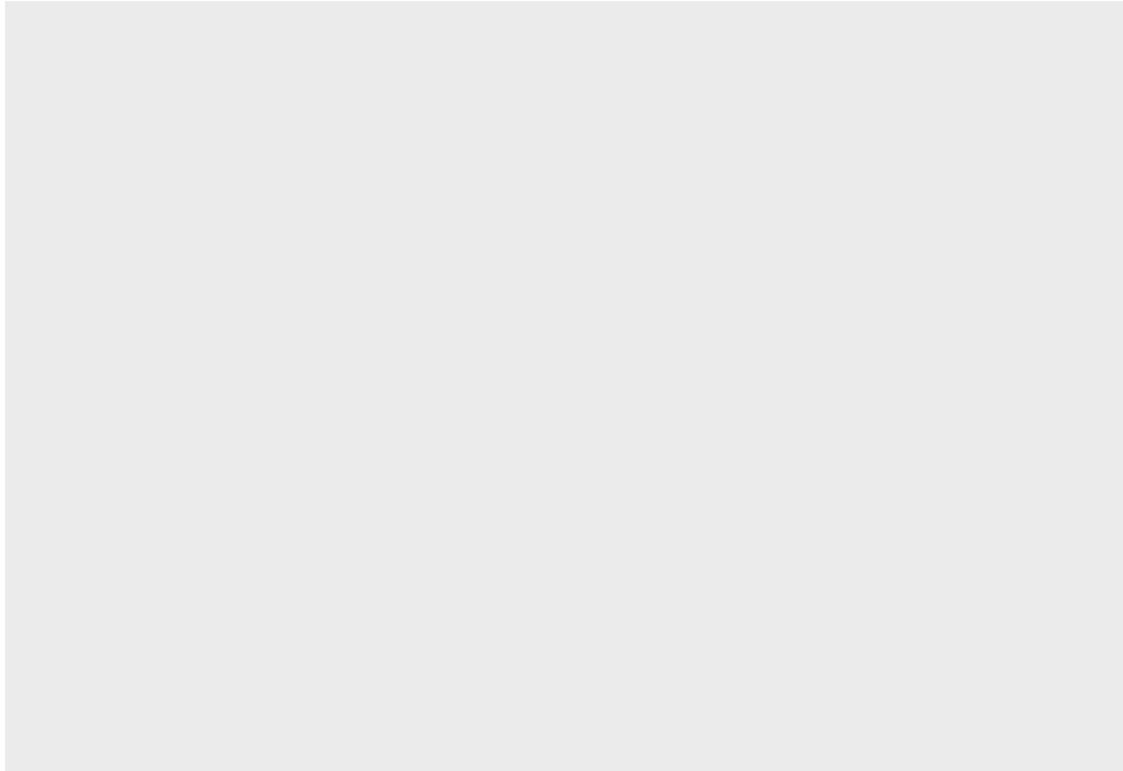
17.614 sample6 KO typeA 3 15.551529 32

```
library(ggplot2)
```

The `ggplot()` function is used to **initialize the basic graph structure**, then we add to it. The basic idea is that you specify different parts of the plot using additional functions one after the other and combine them using the `+` operator; the functions in the resulting code chunk are called layers.

1st layer: plot window

```
ggplot(new_metadata) # what happens? we get a plot window
```



The **geom (geometric) object** is the layer that specifies what kind of plot we want to draw. A plot **must have at least one geom**; there is no upper limit. Examples include:

- points (`geom_point`, `geom_jitter` for scatter plots, dot plots, etc)
- lines (`geom_line`, for time series, trend lines, etc)
- boxplot (`geom_boxplot`, for, well, boxplots!)

2nd layer: geometries

```
ggplot(new_metadata) + geom_point() # note what happens here
```

17.614.1 Geometries

Geoms (e.g.) in layers: - `geom_point()` - `geom_boxplot()` - `geom_bar()` - `geom_density()` - `geom_dotplot()`

17.614.2 Aesthetics

We get an error because each type of geom usually has a required set of aesthetics. “Aesthetics” are set with the `aes()` function and can be set within `geom_point()` or within `ggplot()`.

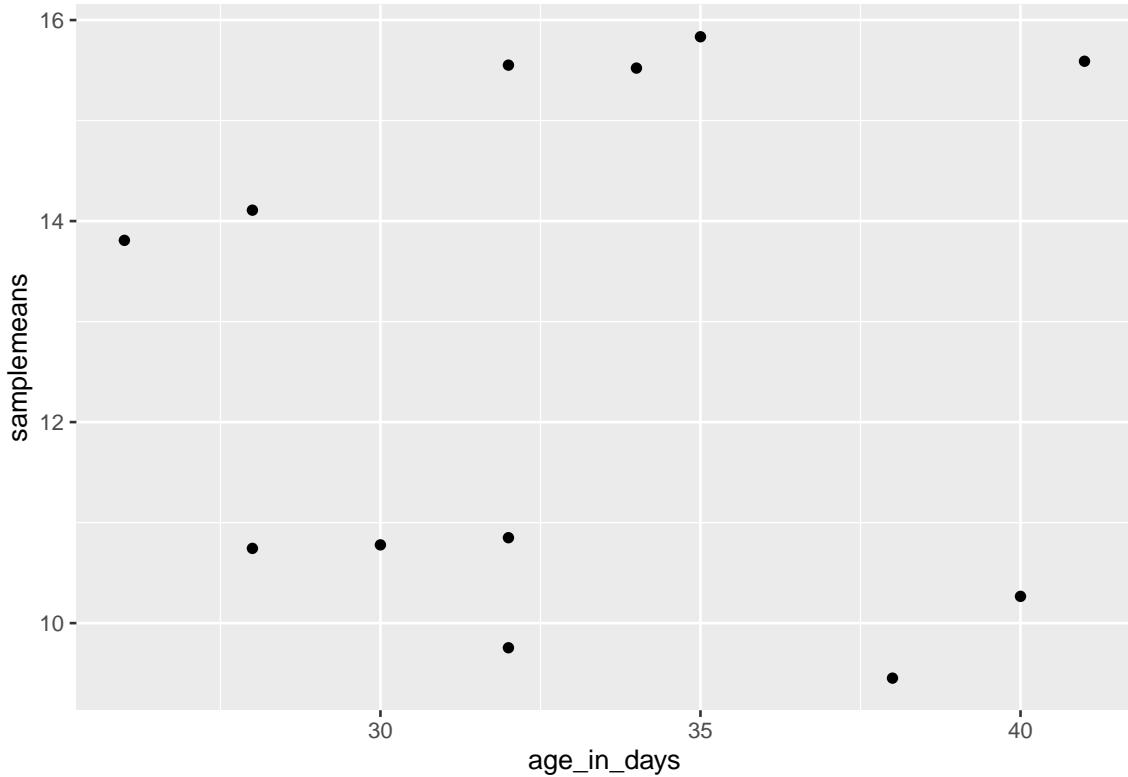
The `aes()` function can be used to specify many plot elements including the following:

- position (i.e., on the x and y axes)

- color (“outside” color)
- fill (“inside” color)
- shape (of points)
- etc.

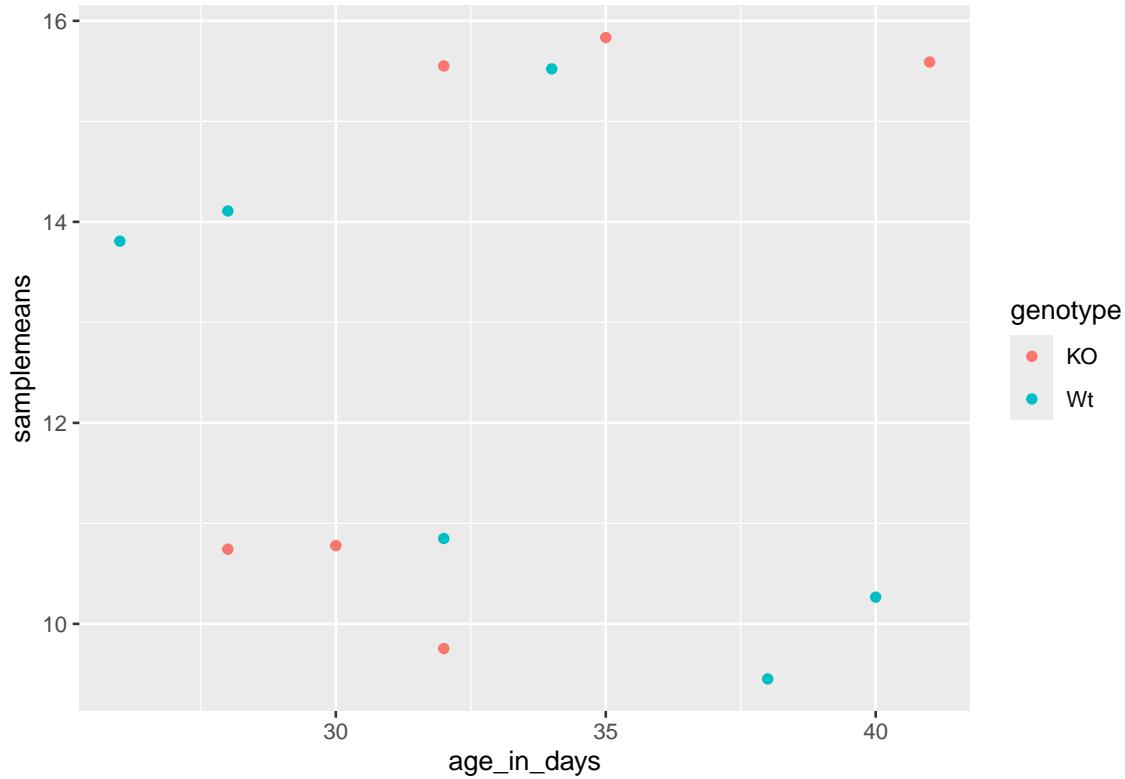
3rd layer: aesthetics

```
ggplot(new_metadata, aes(x = age_in_days, y = samplemeans)) + geom_point() # what happens here? we initialize  
the plot window with the axes
```



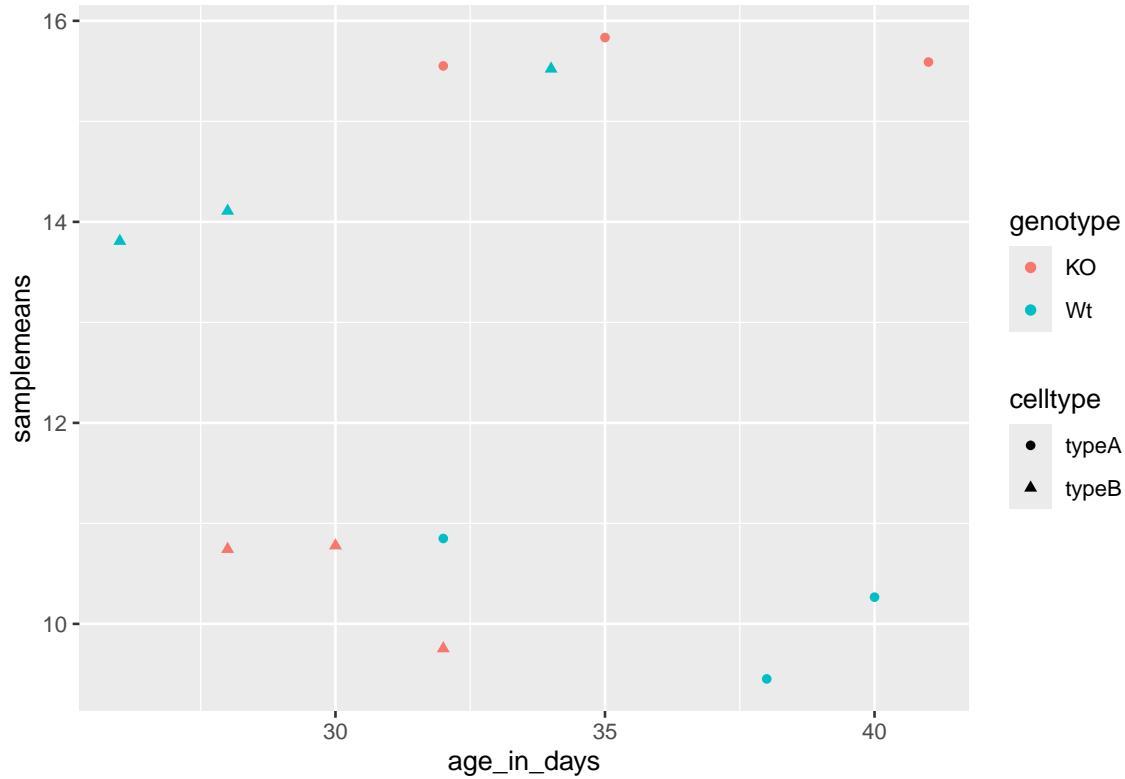
Add color to aesthetics:

```
ggplot(new_metadata) + geom_point(aes(x = age_in_days, y = samplemeans, color = genotype))
```



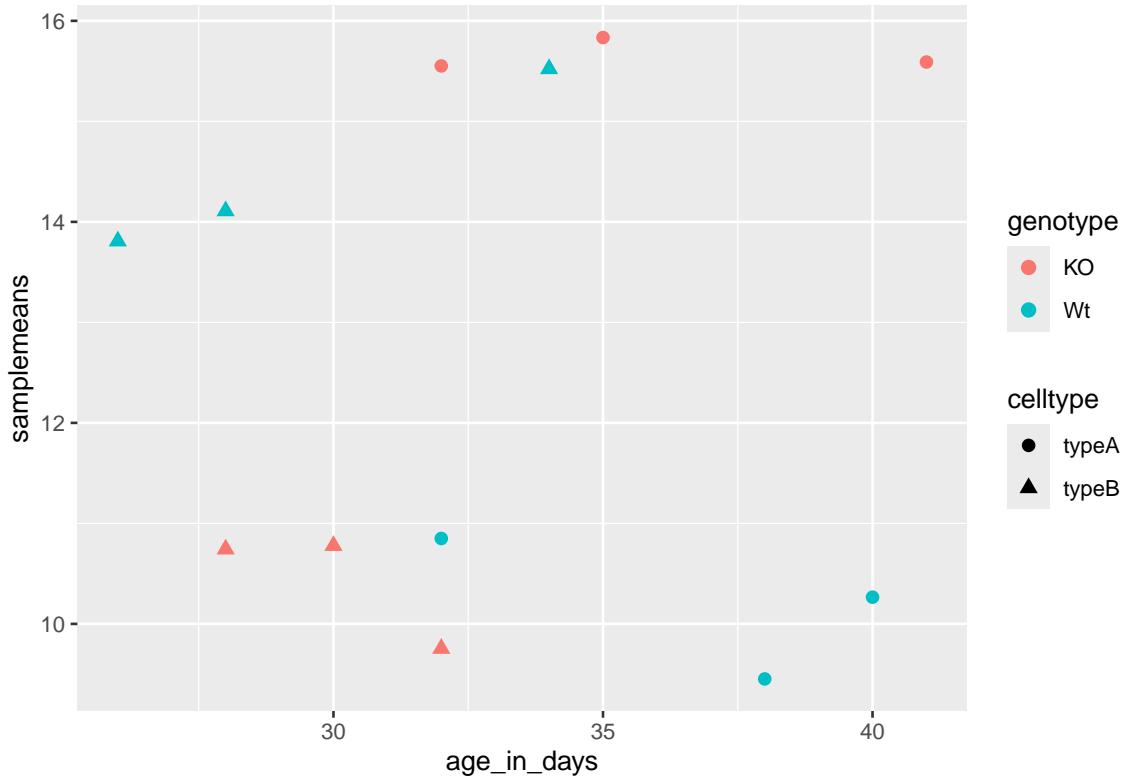
Add shape to aesthetics:

```
ggplot(new_metadata) + geom_point(aes(x = age_in_days, y = samplemeans, color = genotype,  
shape = celltype))
```



Add point size to geometry:

```
ggplot(new_metadata) + geom_point(aes(x = age_in_days, y = samplemeans, color = genotype,  
shape = celltype), size = 2.25)
```

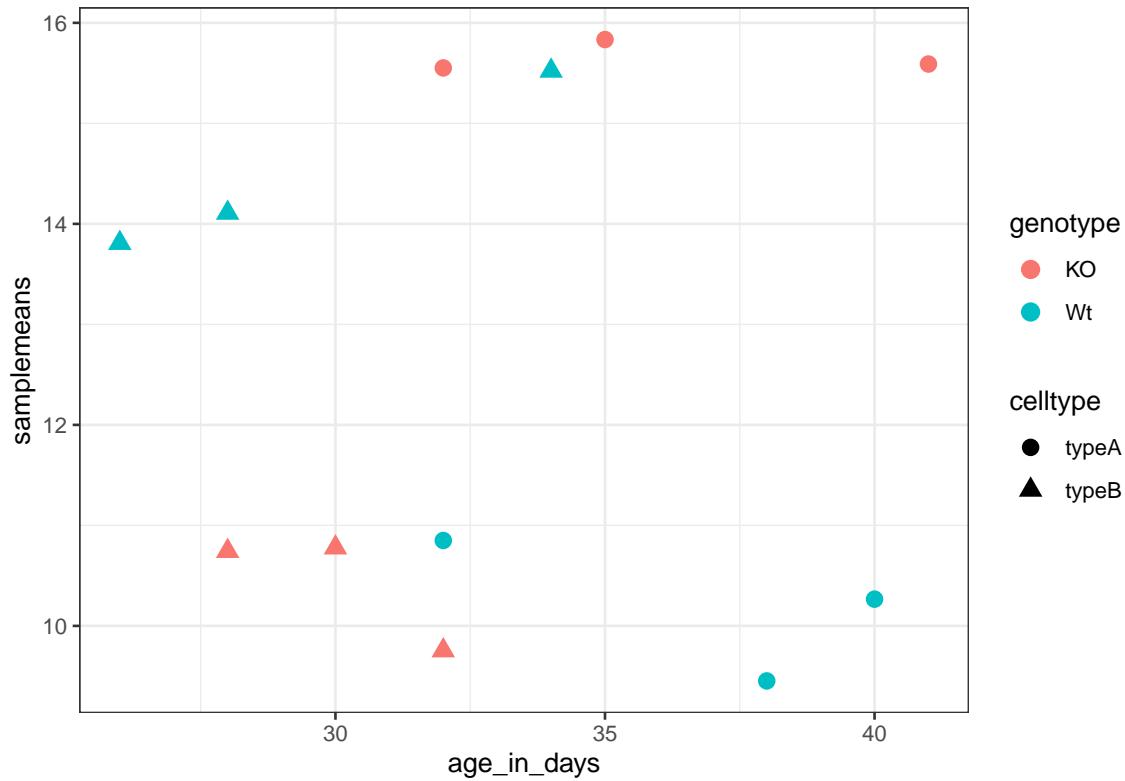


The labels on the x- and y-axis are also quite small and hard to read. To change their size, we need to add an additional theme layer. The ggplot2 theme system handles non-data plot elements such as:

- Axis label aesthetics
- Plot background
- Facet label background
- Legend appearance

5th layer: theme

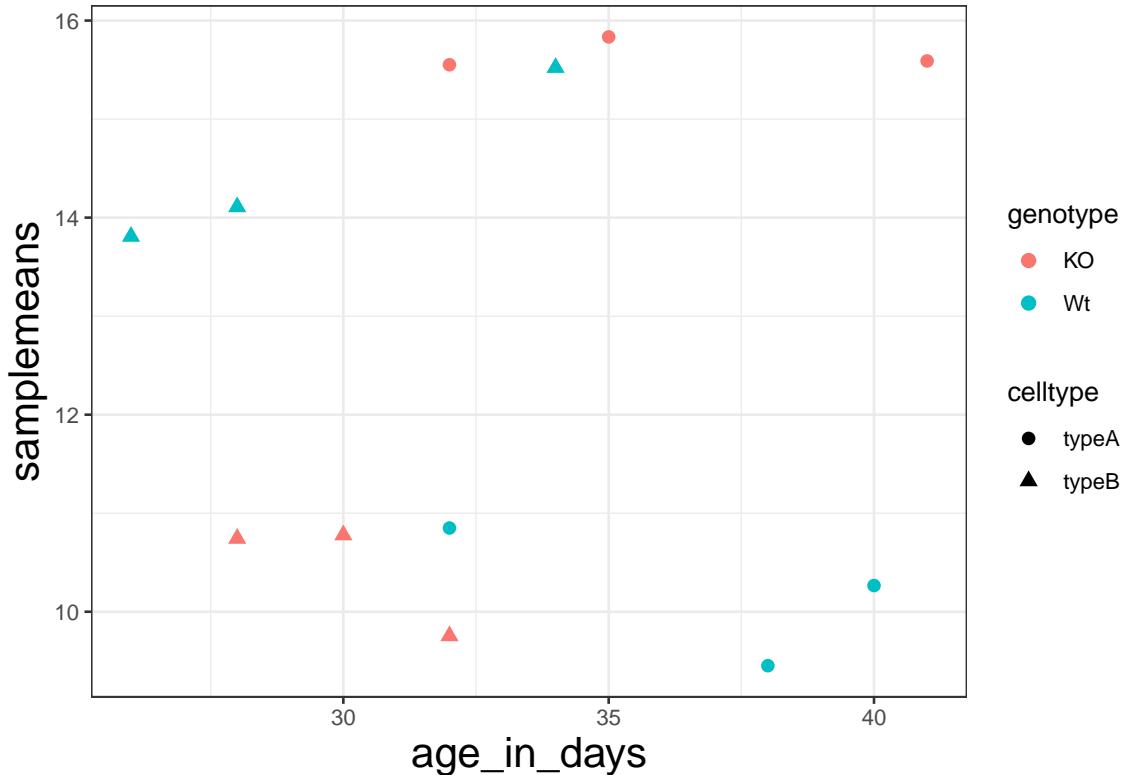
```
ggplot(new_metadata) + geom_point(aes(x = age_in_days, y = samplemeans, color = genotype,
shape = celltype), size = 3) + theme_bw()
```



Do the axis labels or the tick labels get any larger by changing themes?

6th layer: axes title size

```
ggplot(new_metadata) + geom_point(aes(x = age_in_days, y = samplemeans, color = genotype,
shape = celltype), size = 2.25) + theme_bw() + theme(axis.title = element_text(size = rel(1.5))) # increase
the size of the axis titles
```



Exercise points +5

1. The current axis label text defaults to what we gave as input to geom_point (i.e the column headers). We can change this by adding additional layers called **xlab()** and **ylab()** for the x- and y-axis, respectively. Add these layers to the current plot such that the x-axis is labeled “Age (days)” and the y-axis is labeled “Mean expression”.
2. Use the **ggtitle** layer to add a plot title of your choice.
3. Add the following new layer to the code chunk: theme(plot.title = element_text(hjust = 0.5))

```
# Your code here
```

What does it change?

- Ans:

How many theme() layers can be added to a ggplot code chunk, in your estimation?

- Ans:

17.615 13 Boxplot exercise points +16

.md file = 13-boxplot_exercise.md

1. Generate a boxplot using the data in the new_metadata dataframe. Create a ggplot2 code chunk with the following instructions:

- Use the geom_boxplot() layer to plot the differences in sample means between the Wt and KO genotypes.
- Use the fill aesthetic to look at differences in sample means between the celltypes within each genotype.
- Add a title to your plot.
- Add labels, ‘Genotype’ for the x-axis and ‘Mean expression’ for the y-axis.
- Make the following theme() changes:
 - a) Use the theme_bw() function to make the background white.
 - b) Change the size of your axes labels to 1.25x larger than the default.
 - c) Change the size of your plot title to 1.5x larger than default.
 - d) Center the plot title.

1.

```
# Your code here
```

2. Change genotype order: Factor the new_metadata\$genotype column without creating any extra variables/objects and change the levels to c("Wt", "KO")

```
# Your code here
```

3. Re-run the boxplot code chunk you created in 1.

```
# Your code here
```

4. Change default colors

Add a new layer scale_color_manual(values=c("purple","orange")). Do you observe a change?

- Ans:

Replace scale_color_manual(values=c("purple","orange")) with scale_fill_manual(values=c("purple","orange")). Do you observe a change?

- Ans:

In the scatterplot we drew in class, add a new layer scale_color_manual(values=c("purple","orange")), do you observe a difference?

- Ans:

What do you think is the difference between `scale_color_manual()` and `scale_fill_manual()`?

- Ans:
- Boxplot using “color” instead of “fill”. Use purple and orange for your colors.

```
# Your code here
```

- Back in your boxplot code, change the colors in the `scale_fill_manual()` layer to be your 2 favorite colors.

```
# Your code here
```

Are there any colors that you tried that did not work?

- Ans:

5. OPTIONAL Exercise

Find the hexadecimal code for your 2 favourite colors (from exercise 3 above) and replace the color names with the hexadecimal codes within the ggplot2 code chunk.

```
# Hint: the gplots package. If gplots is not already installed, uncomment the
# following line of code

# BiocManager::install('gplots')

library(gplots)
```

```
# Your code here
```

17.616 14 Saving data and plots to file

.md file = 14-exporting_data_and_plots.md

17.616.1 Writing data to file

```
# Save a data frame to file
write.csv(sub_meta, file = "data/subset_meta.csv")
```

17.616.2 Exporting figures to file:

```
# opens the graphics device for writing to a file
pdf(file = "figures/scatterplot.pdf")

ggplot(new_metadata) + geom_point(aes(x = age_in_days, y = samplemeans, color = genotype,
shape = celltype), size = rel(3))

dev.off() # closes the graphics device
```

17.617 pdf

17.618 2

17.619 15 Finding help

.md file = 15-finding_help.md

17.619.1 Saving variables

In order to find help on line you must provide a reproducible example. **Google** and R sites such as **Stackoverflow** are good sources. To provide an example you may want to share an object with someone else, you can save any or all R data structures that you have in your environment to a file.

The function `save.image()` saves all environmental variables in your current R session to a file called “.RData” in your working directory. This is a command that you should use often when in an R session to save your work.

```
save.image()
# OR give it a memorable name
save.image("2024_IntroR_and_RStudio.RData")
# OR for a single object
save.image(metadata, "metadata.RData")
# you can also use
saveRDS(metadata, file = "metadata.RDS") # we'll see how to load this later
```

17.619.2 Loading data

```
load(file = ".RData") # to load .RData files
readRDS(file = "metadata.RDS") # to load .rds files
```

17.619.3 sessionInfo()

Lists details about your current R session: R version and platform; locale and timezone; all packages and versions that are loaded as we saw before.

```
# you'll also often be required to provide your session info
sessionInfo()
```

17.620 16 Tidyverse data wrangling

.md file = 16-tidyverse.md

```
# If tidyverse is not already installed, uncomment the following line of code
# BiocManager::install('tidyverse')

library(tidyverse)
```

Tidyverse basics

17.620.1 Pipes: The pipe (%>%) allows the output of a previous command to be used as input to another command instead of using nested functions.

NOTE: Shortcut to write the pipe is shift + command + m on Macos; shift + ctrl + m on Windows

```
## A single command
sqrt(83)
```

17.621 [1] 9.110434

```
## Base R method of running more than one command
round(sqrt(83), digits = 2)
```

17.622 [1] 9.11

```
## Running more than one command with piping
sqrt(83) %>%
  round(digits = 2)
```

17.623 [1] 9.11

Exercise points +2

1. Create a vector of random numbers using the code below:

```
random_numbers <- c(81, 90, 65, 43, 71, 29)
```

2. Use the pipe (%>%) to perform two steps in a single line:

- a) Take the mean of random_numbers using the mean() function.
- b) Round the output to three digits using the round() function.

```
# Your code here
```

17.623.1 Tibbles

Experimental data

The dataset: gprofiler_results_Mov10oe.tsv

The gene list represents the functional analysis results of differences between control mice and mice over-expressing a gene involved in RNA splicing. We will focus on gene ontology (GO) terms, which describe the roles of genes and gene products organized into three controlled vocabularies/ontologies (domains):

- biological processes (BP)
- cellular components (CC)
- molecular functions (MF)

that are over-represented in our list of genes.

17.623.2 Analysis goal and workflow

Goal: Visually compare the most significant biological processes (BP) based on the number of associated differentially expressed genes (gene ratios) and significance values and create a plot.

1. Read in the functional analysis results

```
# Read in the functional analysis results
functional_GO_results <- read.delim(file = "data/gprofiler_results_Mov10oe.tsv")

# Take a look at the results
functional_GO_results[1:3, 1:12]
```

17.624 query.number significant p.value term.size query.size overlap.size recall

17.625 1 1 TRUE 0.00434 111 5850 52 0.009

17.626 2 1 TRUE 0.00330 110 5850 52 0.009

17.627 3 1 TRUE 0.02970 39 5850 21 0.004

17.628 precision term.id domain subgraph.number

17.629 1 0.468 GO:0032606 BP 237

17.630 2 0.473 GO:0032479 BP 237

17.631 3 0.538 GO:0032480 BP 237

17.632 term.name

17.633 1 type I interferon production

17.634 2 regulation of type I interferon production

17.635 3 negative regulation of type I interferon production

```
names(functional_GO_results)
```

17.636 [1] “query.number” “significant” “p.value” “term.size”

17.637 [5] “query.size” “overlap.size” “recall” “precision”

17.638 [9] “term.id” “domain” “subgraph.number” “term.name”

17.639 [13] “relative.depth” “intersection”

```
names(functional_GO_results)
```

17.640 [1] "query.number" "significant" "p.value" "term.size"

17.641 [5] "query.size" "overlap.size" "recall" "precision"

17.642 [9] "term.id" "domain" "subgraph.number" "term.name"

17.643 [13] "relative.depth" "intersection"

2. Extract only the GO biological processes (BP) of interest

```
# Return only GO biological processes
bp_oe <- functional_GO_results %>%
  dplyr::filter(domain == "BP")
bp_oe[1:3, 1:12]
```

17.644 query.number significant p.value term.size query.size overlap.size recall

17.645 1 1 TRUE 0.00434 111 5850 52 0.009

17.646 2 1 TRUE 0.00330 110 5850 52 0.009

17.647 3 1 TRUE 0.02970 39 5850 21 0.004

17.648 precision term.id domain subgraph.number

17.649 1 0.468 GO:0032606 BP 237

17.650 2 0.473 GO:0032479 BP 237

17.651 3 0.538 GO:0032480 BP 237

17.652 term.name

17.653 1 type I interferon production

17.654 2 regulation of type I interferon production

17.655 3 negative regulation of type I interferon production

Note: the dplyr::filter is necessary because the stats library gets loaded with ggplot2, rmarkdown and bookdown and stats also has a filter function

Exercise points +1

We would like to perform an additional round of filtering to only keep the most specific GO terms.

1. For bp_oe, use the filter() function to only keep those rows where the relative.depth is greater than 4.
2. Save output to overwrite our bp_oe variable.

```
bp_oe <- bp_oe %>%
  dplyr::filter(relative.depth > 4)
```

17.655.1 Select

3. Select only the colnames needed for visualization

```
# Selecting columns to keep for visualization
names(bp_oe)
```

17.656 [1] "query.number" "significant" "p.value" "term.size"

17.657 [5] "query.size" "overlap.size" "recall" "precision"

17.658 [9] "term.id" "domain" "subgraph.number" "term.name"

17.659 [13] "relative.depth" "intersection"

```
bp_oe <- bp_oe %>%
  select(term.id, term.name, p.value, query.size, term.size, overlap.size, intersection)

head(bp_oe[, 1:6])
```

17.660 term.id term.name p.value query.size

17.661 1 GO:0090672 telomerase RNA localization 2.41e-02 5850

17.662 2 GO:0090670 RNA localization to Cajal body 2.41e-02 5850

17.663 3 GO:0090671 telomerase RNA localization to Cajal body 2.41e-02
5850

17.664 4 GO:0071702 organic substance transport 7.90e-11 5850

17.665 5 GO:0015931 nucleobase-containing compound transport 4.43e-05
5850

17.666 6 GO:0050657 nucleic acid transport 3.67e-06 5850

17.667 term.size overlap.size

17.668 1 16 11

17.669 2 16 11

17.670 3 16 11

17.671 4 2629 973

17.672 5 200 93

17.673 6 166 83

```
dim(bp_oe)
```

17.674 [1] 668 7

17.674.1 Arrange

Let's sort the rows by adjusted p-value with the arrange() function.

```
# Order by adjusted p-value ascending
bp_oe <- bp_oe %>%
  arrange(p.value)
```

17.674.2 Rename

Let's rename the term.id and term.name columns.

```
# Provide better names for columns
bp_oe <- bp_oe %>%
  dplyr::rename(GO_id = term.id, GO_term = term.name)
```

Exercise points +1

Rename the intersection column to genes to reflect the fact that these are the DE genes associated with the GO process.

```
# Your code here
```

17.674.3 Mutate

Create additional metrics for plotting (e.g. gene ratios)

```
# Create gene ratio column based on other columns in dataset
bp_oe <- bp_oe %>%
  mutate(gene_ratio = overlap.size/query.size)
```

Exercise points +1

Create a column in bp_oe called term_percent to determine the percent of DE genes associated with the GO term relative to the total number of genes associated with the GO term (overlap.size / term.size)

```
# Your code here
```

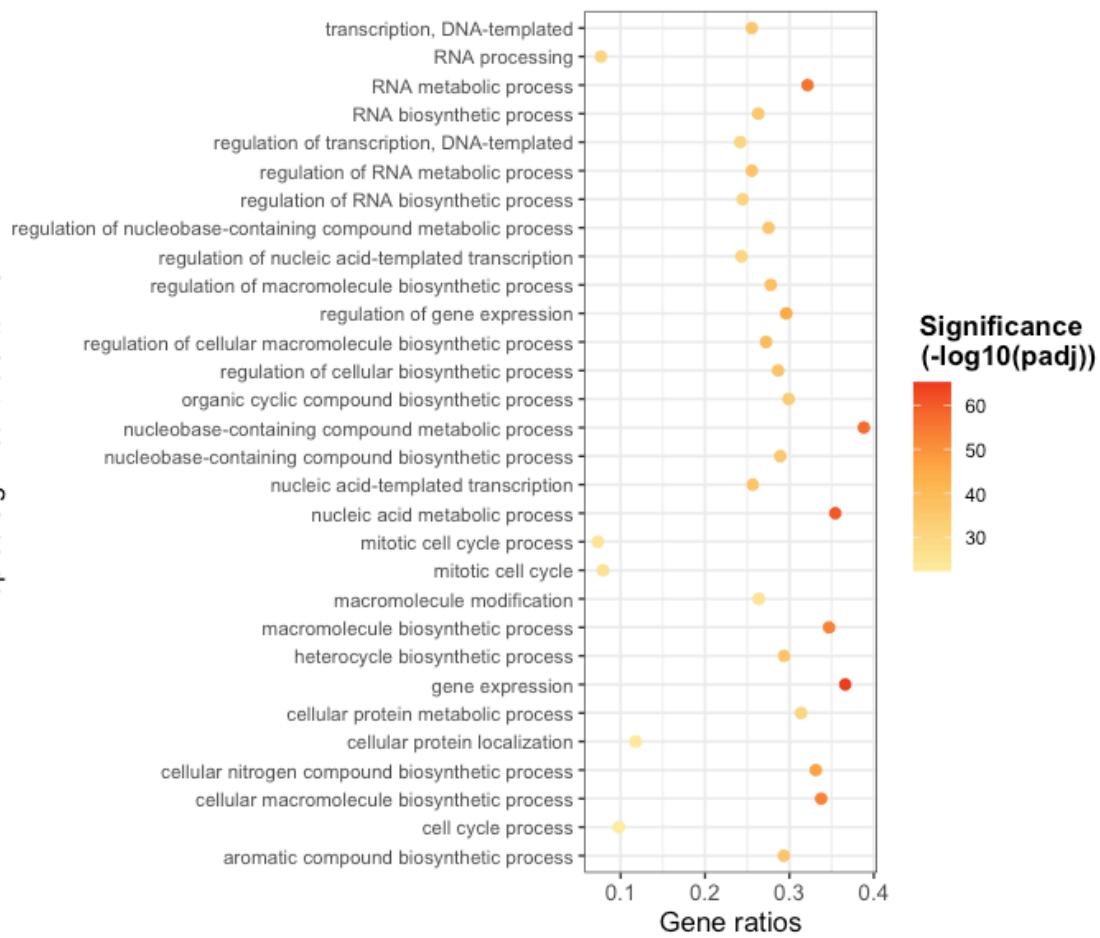
Gives you a sense of how many of the genes in the GO term are differentially expressed.

Exercise points +4

Use ggplot to plot the first 30 significant GO terms vs gene_ratio. Color the points by p.value as shown in the figure in below:

Top 30 significant GO terms

Dotplot of top 30 significant GO terms



```
sub = bp_oe[1:30, ]
```

Change the p.value gradient colors to red-orange-yellow

```
# to help linearize the p.values for the colors; not essential for answer
sub`-log10(p.value)` = -log10(sub$p.value)
# Your code here
```

See ggplot for more detail.