STATS380 SEMESTER 1 2025 | Final Exam

Code Development: Introduction

The Youth Justice Indicator Report

Ram raids have been recently reported to be featured regularly in the media amongst youth near the end of 2022. Although, a couple of sources claim otherwise. We will evaluate this for ourselves.

A Manual Approach

Since we only need cells X20 and AH23, we can simply go to the Excel Workbook, copy and paste the content of these specified cells into a .txt file and use this for our analysis.

Rate per 10,000 Population (adjusted)*										
2010/11	2011/12	2012/13	2013/14	2014/15	2015/16	2016/17	2017/18	2018/19	2019/20	2020/21
489	447	397	333	300	268	249	263	222	205	177
147	134	106	98	78	84	77	46	37	40	34
98	84	76	65	57	46	43	43	33	33	28

↓

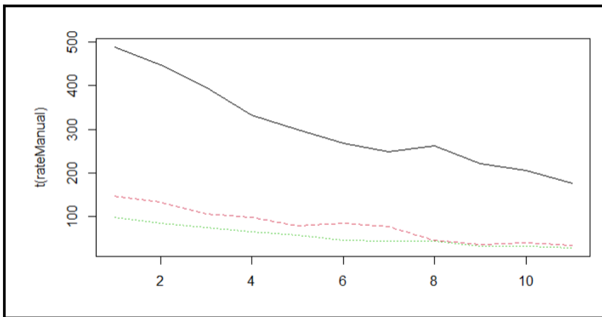
x20_ah23 - Notepad

File Edit Format View Help

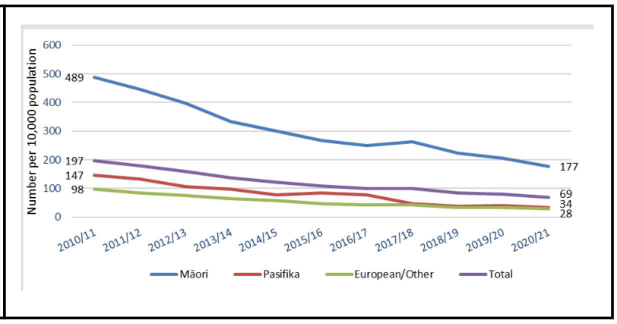
```
2010/11 2011/12 2012/13 2013/14 2014/15 2015/16 2016/17 2017/18 2018/19 2019/20 2020/21
489     447     397     333     300     268     249     263     222     205     177
147     134     106     98      78      84      77      46      37      40      34
98      84      76      65      57      46      43      43      33      33      28
```

```
rateManual <- read.csv('x20_ah23.txt', sep = "\t")
matplot(t(rateManual), type="l")
```

Manual Approach using X20 - AH23



Youth Justice Indicators Summary Report



As seen from the YJI Summary Report vs our Manual Approach, the graphs look similar and the claims appear to hold. Although, the copy and pasted .txt file only includes children of ages 10 to 13, while there is a separate sheet for children ages 14-16. As statisticians, we know better, that other variables may influence our hypothesis. We need to, therefore, dig deeper.

Manual Approach	Code-Based Approach
• No record of what I did (collaboration) • Costly, tedious and error prone • Working with a mouse is limiting	• Provides code record of what I did • Easier to repeat, reproduce, modify, debug • Code is accurate and expressive

Code Development: Functions

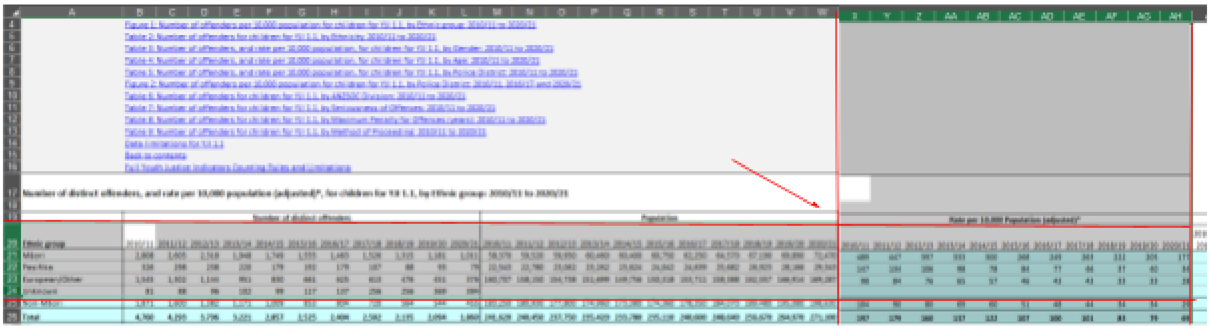
The 'YJI 1.1 Children' and 'YJI 1.1 Young People' sheets are populated with a messy collection of tables, hence we need to take subsets of each sheet.

```
naive <- read.csv("CSV/2021-1.1-Children.csv")
head(naive)
```

```
  YJI.1.1..Offending.rates.per.10.000.population.for.children.aged.10.to.13
1                                                                       <NA>
2                                                               On this page
3                                                                       <NA>
4                                                                       <NA>
5                                                                       <NA>
6                                                                       <NA>

1
2 Table 1: Number of offenders, and rate per 10,000 population, for children for YJI 1.1, by I
3           Figure 1: Number of offenders per 10,000 population for children for YJI 1.1, by I
4                                     Table 2: Number of offenders for children for YJI 1.1, l
5       Table 3: Number of offenders, and rate per 10,000 population, for children for YJI 1.:
6          Table 4: Number of offenders, and rate per 10,000 population, for children for YJI
  ...3 ...4 ...5 ...6 ...7 ...8 ...9 ...10 ...11 ...12 ...13 ...14 ...15 ...16
1 <NA> <NA> <NA> <NA> <NA> <NA> <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
2 <NA> <NA> <NA> <NA> <NA> <NA> <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
```

As seen from the above data, the sheet contains a lot of information we can ignore. Cells X20 to AH23 are what we are actually interested in, hence, we should extract the 20 to 24th rows of the 24th or 34th column.



```
rateCols <- 24:34
rateByEthnicGroup <- read.csv("CSV/2021-1.1-Children.csv", skip=19, nrows=3)[rateCols]
rateByEthnicGroup
```

```
  X2010.11.2 X2011.12.2 X2012.13.2 X2013.14.2 X2014.15.2 X2015.16.2 X2016.17.2
1  489.39727  446.82714  396.68769  332.73327  299.96375  268.40380  249.22280
2  147.11407  133.55241  105.87843   97.66972   78.49106   83.73942   77.03974
3   97.80214   84.04996   75.84956   64.74028   57.41285   46.11012   43.11791
```

```
 X2017.18.2 X2018.19.2 X2019.20.2 X2020.21.2
  263.26996   222.3782  205.12684  176.99938
   46.41263    37.1390   40.08221   34.15912
   43.11369    33.2715   32.79973   28.18008
```

```
yearNames <- 2011:2021
colnames(rateByEthnicGroup) <- yearNames
```

```
       2011      2012      2013      2014      2015      2016      2017
1 489.39727 446.82714 396.68769 332.73327 299.96375 268.40380 249.22280
2 147.11407 133.55241 105.87843  97.66972  78.49106  83.73942  77.03974
3  97.80214  84.04996  75.84956  64.74028  57.41285  46.11012  43.11791
```

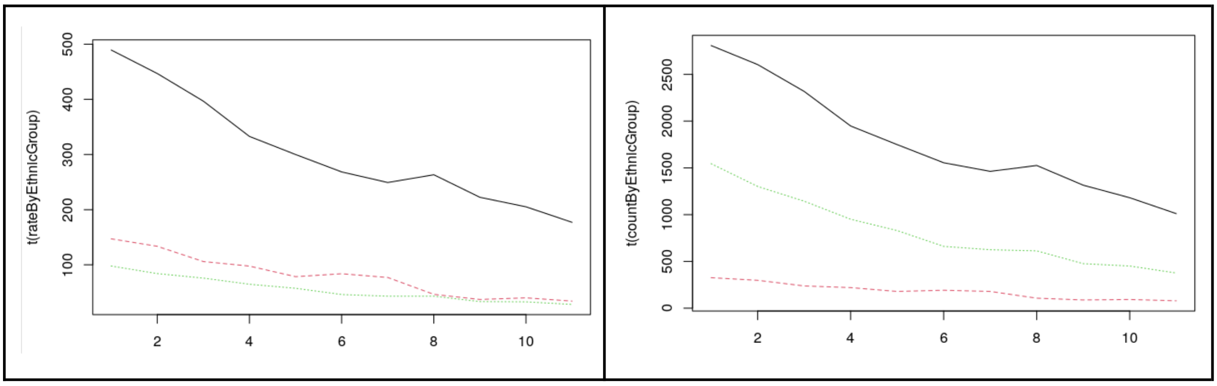
```
     2018     2019      2020      2021
263.26996 222.3782 205.12684 176.99938
 46.41263  37.1390  40.08221  34.15912
 43.11369  33.2715  32.79973  28.18008
```

We have now completely replicated our initial manual approach in a code-based approach, achieving the same outcome. We now also have a record of the process, making it easier to replicate further. For example, if we wanted to extract the Number of Distinct Offenders, two sub-tables to the left of the Rate, we would need the same rows, only changing columns from 24th or 34th, to 2nd to 12th.

```
countCols <- 2:12
countByEthnicGroup <- read.csv(csvFile, skip=20, nrows=3, header=FALSE)[countCols]
colnames(countByEthnicGroup) <- yearNames
```

	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020	2021
1	2808	2605	2318	1948	1749	1555	1463	1526	1315	1181	1011
2	326	298	238	220	179	192	179	107	88	93	79
3	1545	1302	1144	951	830	661	625	613	476	451	376

Rate by Ethnic Group Count by Ethnic Group



Now, to differentiate between the two graphs above, we should use the previous column, before the 2010/11 columns as indicators to label our tables.

```
rateByEthnicGroup <- read.csv(csvFile, skip=20, nrows=3, header=FALSE)[c(1, rateCols)]
colnames(rateByEthnicGroup) <- c("group", yearNames)
```

	group	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020	2021
1	Māori	489.39727	446.82714	396.68769	332.73327	299.96375	268.40380	249.22280	263.26996	222.3782	205.12684	176.99938
2	Pasifika	147.11407	133.55241	105.87843	97.66972	78.49106	83.73942	77.03974	46.41263	37.1390	40.08221	34.15912
3	European/Other	97.80214	84.04996	75.84956	64.74028	57.41285	46.11012	43.11791	43.11369	33.2715	32.79973	28.18008

In the above code, [c(1, rateCols)], is essentially a vector of 1, 24, 25, 26, …, 34. Hence, the code extracts everything below row 19 (but stops at row 23 because nrows=3), for columns 1, 24, …, 34. Moreover, typically, data frames are stored "normalised" or in "long-format", so using reshape2::melt() function in base R makes it simple to do this. Side note, the melt funcion turns 'Year" into a factor, so converting it back to numeric is useful.

```
rateByEthnicGroupLong <- reshape2::melt(rateByEthnicGroup, id.vars="group",
                                        variable="year", value.name="rate")
rateByEthnicGroupLong$year <- as.numeric(as.character(rateByEthnicGroupLong$year))
```

	group	2011	2012	2013	2014	2015	2016
1	Māori	489.39727	446.82714	396.68769	332.73327	299.96375	268.40380
2	Pasifika	147.11407	133.55241	105.87843	97.66972	78.49106	83.73942
3	European/Other	97.80214	84.04996	75.84956	64.74028	57.41285	46.11012

→

	group	year	rate
1	Māori	2011	489.39727
2	Pasifika	2011	147.11407
3	European/Other	2011	97.80214
4	Māori	2012	446.82714
5	Pasifika	2012	133.55241
6	European/Other	2012	84.04996

Functions to Generalise a Solution

The final two complete sets of code are provided below. Although, between the two functions, the only we changed were the column to keep and the name of the variable we were extracting, therefore, we can generalise to create a function as opposed to a block of code. The third example shows how we can combine into a function.

```
rateByEthnicGroup <- read.csv(csvFile, skip=20, nrows=3,header=FALSE)[c(1, rateCols)]
colnames(rateByEthnicGroup) <- c("group", yearNames)
rateByEthnicGroupLong <-  reshape2::melt(rateByEthnicGroup, id.vars="group",
                             variable="year", value.name="rate")
rateByEthnicGroupLong$year <- as.numeric(as.character(rateByEthnicGroupLong$year))
```

```
countByEthnicGroup <- read.csv(csvFile, skip=20, nrows=3,header=FALSE)[c(1, countCols)]
colnames(countByEthnicGroup) <- c("group", yearNames)
countByEthnicGroupLong <- reshape2::melt(countByEthnicGroup, id.vars="group",
                             variable="year", value.name="count")
countByEthnicGroupLong$year <- as.numeric(as.character(countByEthnicGroupLong$year))
```

```
tableByEthnicGroup <- function(keep, name) {
  cols <- c(1, keep)
  df <- read.csv(csvFile, skip=20, nrows=3, header=FALSE)[cols]
  colnames(df) <- c("group", yearNames)
  dfLong <- reshape2::melt(df, id.vars="group",  variable="year", value.name=name)
  dfLong$year <- as.numeric(as.character(dfLong$year))
  dfLong
}
```

```
rateByEthnicGroupLong <- tableByEthnicGroup(rateCols, "rate")
countByEthnicGroupLong <- tableByEthnicGroup(countCols, "count")
popByEthnicGroupLong <- tableByEthnicGroup(popCols, "pop")
```

Merging Tables

If we wanted to merge the rate and count into a single dataframe we use merge(), alternatively, to combine all three, count, population and rate, we could use cbind()

```
childrenByEthnicGroup <- merge(rateByEthnicGroupLong, countByEthnicGroupLong,
                          by=c("year", "group"))

childrenByEthnicGroup <- cbind(countByEthnicGroupLong, popByEthnicGroupLong[3],
                          rateByEthnicGroupLong[3])
```

merge()

	year	group	rate	count
1	2011	European/Other	97.80214	1545
2	2011	Māori	489.39727	2808
3	2011	Pasifika	147.11407	326
4	2012	European/Other	84.04996	1302
5	2012	Māori	446.82714	2605
6	2012	Pasifika	133.55241	298

cbind()

	group	year	count	pop	rate
1	Māori	2011	2808	58369.97	489.39727
2	Pasifika	2011	326	22543.29	147.11407
3	European/Other	2011	1545	160706.72	97.80214
4	Māori	2012	2605	59520.02	446.82714
5	Pasifika	2012	298	22780.30	133.55241
6	European/Other	2012	1302	158149.70	84.04996

Fixed Constants

After combining the three variables, population, count and rate for each ethnicity in the above cbind(), we notice that the Excel workbook contains more tables we can work with. For example, the Number of Distinct Offenders and the Percent of Total (a proportion). We could use our final function tableByEthnicGroup(), although it contains fixed constants, for example see the comparison below. This indicates we can generalise our function further, but this means more parameters are required for the function input.

```r
tableByEthnicGroup <- function(keep, name) {
    cols <- c(1, keep)
    df <- read.csv(csvFile, skip=20, nrows=3,
                   header=FALSE)[cols]
    colnames(df) <- c("group", yearNames)
    dfLong <- reshape2::melt(df, id.vars="group",
                             variable="year", value.name=name)
    dfLong$year <- as.numeric(as.character(dfLong$year))
    dfLong
}
```

```r
getTable <- function(skip, nrows, keep, name, by) {
    cols <- c(1, keep)
    df <- read.csv(csvFile, skip=skip, nrows=nrows,
                   header=FALSE)[cols]
    colnames(df) <- c(by, yearNames)
    dfLong <- reshape2::melt(df, id.vars=by,
                             variable="year", value.name=name)
    dfLong$year <- as.numeric(as.character(dfLong$year))
    dfLong
}
```

Writing Higher-Level Functions

Looking further down the Excel workbook, we see the rate, count and population broken down by gender. If we use our getTable() function defined above, we could skip 75 rows, extract only 2 rows and specify the other parameters as shown below.

	Number of distinct offenders										
Gender	2010/11	2011/12	2012/13	2013/14	2014/15	2015/16	2016/17	2017/18	2018/19	2019/20	2020/21
Male	3,368	3,057	2,696	2,249	2,066	1,839	1,793	1,822	1,514	1,484	1,305
Female	1,391	1,235	1,099	970	790	684	610	678	619	609	550
Other/Unknown	1	1	1	2	1	2	1	2	2	1	5
Total	**4,760**	**4,293**	**3,796**	**3,221**	**2,857**	**2,525**	**2,404**	**2,502**	**2,135**	**2,094**	**1,860**

Population										
2010/11	2011/12	2012/13	2013/14	2014/15	2015/16	2016/17	2017/18	2018/19	2019/20	2020/21
123,990	123,250	121,750	120,500	119,640	120,630	123,330	127,470	131,790	136,260	139,470
117,630	117,190	116,010	114,920	114,140	114,480	117,270	121,180	124,880	128,720	131,630
241,620	**240,450**	**237,750**	**235,420**	**233,780**	**235,110**	**240,600**	**248,640**	**256,670**	**264,970**	**271,100**

Rate per 10,000 Population										
2010/11	2011/12	2012/13	2013/14	2014/15	2015/16	2016/17	2017/18	2018/19	2019/20	2020/21
272	248	221	187	173	152	145	143	115	109	94
118	105	95	84	69	60	52	56	50	47	42
197	**179**	**160**	**137**	**122**	**107**	**100**	**101**	**83**	**79**	**69**

```r
rateByGenderLong <- getTable(75, 2, rateCols, "rate", "gender")
countByGenderLong <- getTable(75, 2, countCols, "count", "gender")
popByGenderLong <- getTable(75, 2, popCols, "pop", "gender")
childrenByGender <- cbind(countByGenderLong, popByGenderLong[3], rateByGenderLong[3])
```

```
  gender year count    pop      rate
1   Male 2011  3368 123990 271.63481
2 Female 2011  1391 117630 118.25215
3   Male 2012  3057 123250 248.03245
4 Female 2012  1235 117190 105.38442
5   Male 2013  2696 121750 221.43737
6 Female 2013  1099 116010  94.73321
```

```
getRateBy <- function(skip, nrows, by) {
  rate <- getTable(skip, nrows, rateCols, "rate", by)
  count <- getTable(skip, nrows, countCols, "count", by)
  pop <- getTable(skip, nrows, popCols, "pop", by)
  cbind(count, pop[3], rate[3])
}
getPropBy <- function(skip, nrows, by) {
  number <- getTable(skip, nrows, numberCols, "number", by)
  prop <- getTable(skip, nrows, propCols, "prop", by)
  cbind(number, prop[3])
}
childrenByAge <- getRateBy(85, 4, "age")
childrenByEthnicGroup <- getRateBy(20, 3, "group")
childrenByGender <- getRateBy(75, 3, "gender")
childrenByEthnicity <- getPropBy(61, 7, "ethnicity")
```

Object	What it contains	Sheet region read
childrenByAge	Counts, population, rates by age	rows 85-88
childrenByEthnicGroup	Counts, population, rates by ethnic group	rows 20-22
childrenByGender	Counts, population, rates by gender	rows 75-77
childrenByEthnicity	Numbers and proportions by ethnicity	rows 61-67

Code Development: Style

Indenting

f <- function(x) {
    expr1
    expr2
    expr3
}
	for (i in values) {
	    expr1
	    expr2
	    expr3
	}
		if (condition) {
		    expr1
		    expr2
		    expr3
		}

Whitespace

```
if (condition) {
        call(a = 1, b = 2, c = 3)
}
```

Break Long Lines

```
call(a=1, b=2, c=3, d=4, e=5, f=6, g=7, h=8, i=9, j=10, k=11, l=12, m=13, n=14, o=15, p=16, q=17)
call(a=1, b=2, c=3, d=4, e=5, f=6, g=7, h=8, i=9, j=10, k=11, l=12,
    m=13, n=14, o=15, p=16, q=17
```

Comments

- Code should have comments
- Every function should comment its purpose, arguments and return value
- Constants should be explained
- Comment assumptions or any limits to generality of the code

Checking Assumptions

Along with comments to explain assumptions, there should be code to check such assumptions.

```
rateByEthnicGroup <- getTable(20, 3, 24:34, "rate", "group")
```

```
            group year      rate
1           Māori 2011 489.39727
2        Pasifika 2011 147.11407
3  European/Other 2011  97.80214
4           Māori 2012 446.82714
5        Pasifika 2012 133.55241
6  European/Other 2012  84.04996
```

In the above, the assumptions are that the getTable() function assumes the first three arguments are numbers and the last two are character values.

```
getTable <- function(skip, nrows, keep, name, by) {
  if (!(is.numeric(skip) && is.numeric(nrows) && is.numeric(keep) &&
       is.character(name) && is.character(by)))
      stop("'skip', 'nrows', and 'keep' must be numeric and 'name' and 'by' must be character")
  ...
 ...
}
```

The stop() function throws a meaningful error and halts execution, used for error validation.

```
getTable <- function(skip, nrows, keep, name, by) {
  cols <- c(1, keep)
  df <- read.csv(csvFile, skip=skip, nrows=nrows, header=FALSE)[cols]
  colnames(df) <- c(by, yearNames)
  dfLong <- reshape2::melt(df, id.vars=by,
                           variable="year", value.name=name)
  dfLong$year <- as.numeric(as.character(dfLong$year))
}
rateByEthnicGroup <- getTable(20, 3, 24, "rate", "group")
```

Looking at the code, keep = 24. So cols = x(1, 24). Therefore, in the following line, when reading in the CSV, it subsets column 1 and column 24.

```
              V1       V24
1          Māori 489.39727
2       Pasifika 147.11407
3 European/Other  97.80214
```

Then, we try to set the column names in this line: `colnames(df) <- c(by, yearNames)`

```
Error in names(x) <- value: 'names' attribute [12] must be the same length as the vector [2]
```

The problem is we have 2 column names V1 and V24 when it's looking for 12, hence trying to assign column names when only 2 column names are given. This means we assumed that the keep argument is the same length as years. we can adjust the getTable() function accordingly.

```
  if (length(keep) != length(yearNames))
      stop("'keep' and 'yearNames' must have the same length")
```

This does not prevent the error, but at least we know what is going wrong when the function runs into this error.

```
Error in getTable(20, 3, 24, "rate", "group"): 'keep' and 'yearNames' must have the same length
```

Global Variables

Ideally, we should avoid using global variables. Instead, make them local variables within a function or make them as parameter input to the function.

Scope and the Search Path

In a function, R first searches local variables (within a function), then global variables, which can be found using ls() command, and finally the loaded packages in the global workspace, found using the search() command. Self-contained functions are easier to debug, reuse and test as they don't depend on an external state, therefore, it's best to limit hidden dependencies by passing everything your function needs to the best of your ability.

Modular Functions

- A function should perform a single, well-defined task.
  - Avoid too difficult to write a comment that describes its purpose
  - Avoid a function with too many arguments.
- The behaviour of a function should only depend on its arguments.
  - Avoid Global Variables
- A return value should be consistent, the result should always be the same data structure and the same inputs should always produce the same output.
  - sapply() is NOT consistent as sometimes it outputs a vector, matrix or list

Code Development: Refactoring

```
getRateBy <- function(skip, nrows, by) {
    rate <- getTable(skip, nrows, rateCols, "rate", by)
    count <- getTable(skip, nrows, countCols, "count", by)
    pop <- getTable(skip, nrows, popCols, "pop", by)
    cbind(count, pop[3], rate[3])
}
```

```
getPropBy <- function(skip, nrows, by) {
    number <- getTable(skip, nrows, numberCols, "number", by)
    prop <- getTable(skip, nrows, propCols, "prop", by)
    cbind(number, prop[3])
}
```

```
getTable <- function(csvFile, skip, nrows, keep, yearNames, name, by) {
    cols <- c(1, keep)
    df <- read.csv(csvFile, skip=skip, nrows=nrows,  header=FALSE)[cols]
    colnames(df) <- c(by, yearNames)
    dfLong <- reshape2::melt(df, id.vars=by, variable="year", value.name=name)
    dfLong$year <- as.numeric(as.character(dfLong$year))
    dfLong
}
```

↓

```
## Extract three tables (counts, populations, and rates)
##
## 'csv' is the name of the CSV file to read.
## 'skip' is the number of rows to ignore.
## 'nrows' is the number of rows to read.
## 'years' are the names for the year columns.
## 'by' is the label for the variable in the first column.
##
## Returns the combined contents of the three tables
## as a data frame in long form.
getRateBy <- function(csv, skip, nrows,
                      years, by) {
   nYears <- length(years)
   countCols <- 1:nYears + 1
   popCols <- countCols + nYears
   rateCols <- popCols + nYears
   rate <- getTable(csv, skip, nrows, rateCols, years, "rate", by)
   count <- getTable(csv, skip, nrows, countCols, years, "count", by)
   pop <- getTable(csv, skip, nrows, popCols, years, "pop", by)
   cbind(count, pop[3], rate[3])
}
```

```
## Extract two tables (numbers and proportions)
##
## 'csv' is the name of the CSV file to read.
## 'skip' is the number of rows to ignore.
## 'nrows' is the number of rows to read.
## 'years' are the names for the year columns.
## 'by' is the label for the variable in the first column.
##
## Returns the combined contents of the two tables
## as a data frame in long form.
getPropBy <- function(csv, skip, nrows,
                      years, by) {
   nYears <- length(years)
   numberCols <- 1:nYears + 1
   propCols <- numberCols + nYears
   number <- getTable(csv, skip, nrows, numberCols, years,
"number", by)
   prop <- getTable(csv, skip, nrows, propCols, years, "prop", by)
   cbind(number, prop[3])
}
```

```
## Extract a single table that consists of a contiguous set
## of columns and a contiguous set of rows.
##
## 'csvFile' is the CSV file to read from.
## 'skip' is the number of rows to ignore.
## 'nrows' is the number of rows to read.
## 'keep' are the columns to keep.
## 'yearNames' are the labels for the years (the columns other than the first).
## 'name' is the label for the variable within the table.
## 'by' is the label for the variable in the first column.
##
## Returns the contents of the table as a data frame in long form.
getTable <- function(csvFile, skip, nrows, keep, yearNames, name, by) {
   cols <- c(1, keep)
   df <- read.csv(csvFile, skip=skip, nrows=nrows,
                  header=FALSE)[cols]
   colnames(df) <- c(by, yearNames)
   dfLong <- reshape2::melt(df, id.vars=by,
                            variable="year", value.name=name)
   dfLong$year <- as.numeric(as.character(dfLong$year))
   dfLong
}
```

The Difference

```
getRateBy <- function(skip, nrows, by) {
  rate <- getTable(skip, nrows, rateCols, "rate", by)
  count <- getTable(skip, nrows, countCols, "count", by)
  pop <- getTable(skip, nrows, popCols, "pop", by)
  cbind(count, pop[3], rate[3])
}
```

→

```
## COMMENTS
getRateBy <- function(csv, skip, nrows,
                      years, by) {
  nYears <- length(years)
  countCols <- 1:nYears + 1
  popCols <- countCols + nYears
  rateCols <- popCols + nYears
  rate <- getTable(csv, skip, nrows, rateCols, years, "rate", by)
  count <- getTable(csv, skip, nrows, countCols, years, "count", by)
  pop <- getTable(csv, skip, nrows, popCols, years, "pop", by)
  cbind(count, pop[3], rate[3])
}
```

```
getPropBy <- function(skip, nrows, by) {
  number <- getTable(skip, nrows, numberCols, "number", by)
  prop <- getTable(skip, nrows, propCols, "prop", by)
  cbind(number, prop[3])
}
```

→

```
## COMMENTS
getPropBy <- function(csv, skip, nrows,
                      years, by) {
  nYears <- length(years)
  numberCols <- 1:nYears + 1
  propCols <- numberCols + nYears
  number <- getTable(csv, skip, nrows, numberCols, years, "number", by)
  prop <- getTable(csv, skip, nrows, propCols, years, "prop", by)
  cbind(number, prop[3])
}
```

```
getTable <- function(csvFile, skip, nrows, keep, yearNames, name, by) {
  cols <- c(1, keep)
  df <- read.csv(csvFile, skip=skip, nrows=nrows,  header=FALSE)[cols]
  colnames(df) <- c(by, yearNames)
  dfLong <- reshape2::melt(df, id.vars=by, variable="year", value.name=name)
  dfLong$year <- as.numeric(as.character(dfLong$year))
  dfLong
}
```

→

```
## COMMENTS
getTable <- function(csvFile, skip, nrows, keep,
                     yearNames, name, by) {
  cols <- c(1, keep)
  df <- read.csv(csvFile, skip=skip, nrows=nrows,
                 header=FALSE)[cols]
  colnames(df) <- c(by, yearNames)
  dfLong <- reshape2::melt(df, id.vars=by,
                           variable="year", value.name=name)
  dfLong$year <- as.numeric(as.character(dfLong$year))
  dfLong
}
```

Code Development: Divide & Conquer

Our functions, getRateBy() and getPropBy() both have degrees of manual approaches, for example, to extract the offending rate or proportion by ethnic group means manually scrolling through the Excel Workbook to count the specified rows and knowing how many rows to skip. We should instead write code to inspect the workbook sheet and determine which rows contain tables of data. That task seems far too big and complex to conduct, hence why we should Divide and Conquer by breaking down the complex problem into smaller chunks.

```
childrenByEthnicGroup <- getRateBy(csvFile, 20, 3, yearNames, "group")
childrenByEthnicity <- getPropBy(csvFile, 61, 7, yearNames, "ethnicity")
```

Building Simple from Complex

First Simplification: For each table, find one table
Second Simplification: For each table, find the table start and find the table end
For the second simplification, first we need to identify existing structure in the worksheet, something that differentiates the tables. We see that the first 30 values in the first column of the CSV are irrelevant, but a table starts where there is one NA value, followed by a NON-NA value.

```
firstColumn <- read.csv(csvFile, header=FALSE)[[1]]
firstColumn[1:30]
```

```
 [1] "YJI 1.1. Offending rates per 10,000 population for children aged 10 to 13"
 [2] NA
 [3] "On this page"
 [4] NA
 [5] NA
 [6] NA
 [7] NA
 [8] NA
 [9] NA
[10] NA
[11] NA
[12] NA
[13] NA
[14] NA
[15] NA
[16] NA
[17] "Number of distinct offenders, and rate per 10,000 population (adjusted)*, for children f
[18] NA
[19] "Ethnic group"
[20] NA
[21] "Māori"
[22] "Pasifika"
[23] "European/Other"
[24] "Unknown"
[25] "Non-Māori"
[26] "Total"
[27] "Ratio (Māori to Pasifika)"
[28] "Ratio (Māori to European/Other)"
[29] "Ratio (Māori to Non-Māori)"
[30] "*Considerable growth in the extent to which ethnicity is not recorded has necessitated e
```

Third Simplification: For each table, for each row, if the next two rows aren't empty, this is the table start, and we should then find the end of the table.

```
for (i in 1:21) {
  if (!is.na(firstColumn[i])) {
    if (!is.na(firstColumn[i + 1])) {
      cat("table starts on row", i, "\n")
    }}}
table starts on row 21
```

We have now identified where a table starts. Although, if we run this code for 1 in 1:24 instead of 1 in 1:21, we get the following output. Our solution does not yet work for the larger problem.

```
table starts on row 21
table starts on row 22
table starts on row 23
table starts on row 24
```

Building Complex from Simple

Now that we have identified where the table starts, by the above code definition, a table starts when there are two non-NA values, which isn't true since the table contents will be NON-NA values. Therefore, we can use logical vectors to hold the stats we are in, that, if we are inTable or not.

```
inTable <- FALSE
for (i in 1:24) {
  if (!inTable && !is.na(firstColumn[i])) {
    if (!is.na(firstColumn[i + 1])) {
      cat("table starts on row", i, "\n")
      inTable <- TRUE
    }}}
table starts on row 21
```

Now we need to find out where the table ends. Looking at the data, it appears that a table ends when there is a row containing "Total".

```
inTable <- FALSE
for (i in 1:30) {
  if (!inTable && !is.na(firstColumn[i]) &&
    !grepl("Ratio", firstColumn[i])) {
    if (!is.na(firstColumn[i + 1])) {
      cat("table starts on row", i, "\n")
      inTable <- TRUE
    }
  }
  if (inTable && firstColumn[i] == "Total") {
    cat("table ends on row", i, "\n")
  }}
table starts on row 21
table ends on row 26
```

Now that our code is starting to look more complex, we should break down into smaller functions.

tableStart <- function(column, inTable, i) {
  !inTable &&
  !is.na(column[i]) &&
  !grepl("^Ratio", column[i]) &&
  !is.na(column[i + 1])
}
	tableEnd <- function(column, inTable, i) {
	  inTable && column[i] == "Total"
	}
		inTable <- FALSE
		for (i in 1:30) {
		  if (**tableStart**(firstColumn, inTable, i)) {
		    cat("table starts on row", i, "\n")
		    inTable <- TRUE
		  }
		  if (**tableEnd**(firstColumn, inTable, i)) {
		    cat("table ends on row", i, "\n")
		  }
		}

Now we can keep a record of the location of all the tables in the workbook sheet, we have the skip and nrows values for all tables.

```
inTable <- FALSE
tableSkip <- numeric()
tableRows <- numeric()
numTables <- 0
for (i in 1:length(firstColumn)) {
  if (tableStart(firstColumn, inTable, i)) {
    numTables <- numTables + 1
    tableSkip[numTables] <- i - 1
    inTable <- TRUE
  }
  if (tableEnd(firstColumn, inTable, i)) {
    tableRows[numTables] <- i - tableSkip[numTables] - 1
    inTable <- FALSE
  }
}
```

```
tableSkip
[1]  20  61  75  85  96 141 164 176 188 196
tableRows
[1]  5  7  3  4 12 16  5  5  5
```

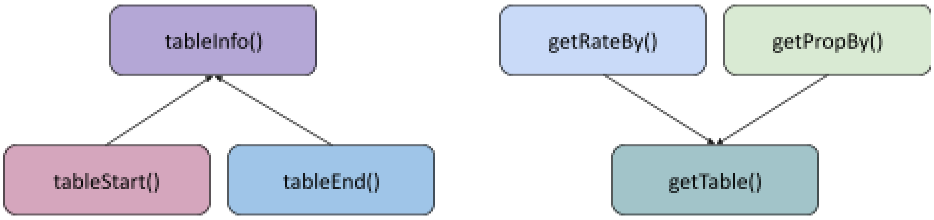
Now we should wrap this entire logic into a function.

```
tableInfo <- function(csvFile) {
  csv <- read.csv(csvFile, header=FALSE)
  column <- csv[,1]
  inTable <- FALSE
  tableSkip <- numeric()
  tableRows <- numeric()
  numTables <- 0
  for (i in 1:length(column)) {
    if (tableStart(column, inTable, i)) {
      numTables <- numTables + 1
      tableSkip[numTables] <- i - 1
      inTable <- TRUE
    }
    if (tableEnd(column, inTable, i)) {
      tableRows[numTables] <- i - tableSkip[numTables] - 1
      inTable <- FALSE
    }
  }
  list(skip=tableSkip, nrows=tableRows)
}
```

```
tableInfo(csvFile)

$skip
[1]  20  61  75  85  96 141 164 176 188 196
$nrows
[1]  5  7  3  4 12 16  5  5  5
```

Code Development: Version Control




```
childrenByEthnicGroup <- getRateBy(csvFile, info$skip[1], info$nrows[1], yearNames, "group")
childrenByEthnicity <- getPropBy(csvFile, info$skip[2], info$nrows[2], yearNames, "ethnicity")
```

```
           group year count       pop      rate
1          Māori 2011  2808  58369.97 489.39727
2       Pasifika 2011   326  22543.29 147.11407
3 European/Other 2011  1545 160706.72  97.80214
4        Unknown 2011    81        NA        NA
5      Non-Māori 2011  1871 183250.01 103.86846
6          Māori 2012  2605  59520.02 446.82714
```

```
  ethnicity year number       prop
1     Māori 2011   2808 60.0128233
2  Pasifika 2011    326  6.9673007
3     Asian 2011     72  1.5387903
4     MELAA 2011     24  0.5129301
5     Other 2011     17  0.3633255
6  European 2011   1432 30.6048301
```

When code works well but it needs some changes, we should implement version control to keep histories of different versions of code. Now we want to repeat the above process for all the tables in the workbook sheet, however there is still some manual intervention required.

- Call getRateBy() or getPropBy() depending on whether the table has counts or proportions
- Explicitly provide the by label ("group" or "ethnicity")
- Explicitly specify the yearNames

Assume we have **REFACTORED** the tableInfo() function to now return the by, type and years.

```
$skip
[1]  20  61  75  85  96 141 164 176 188

$nrows
[1]  5  7  3  4 12 16  5  5  5

$by
[1] "Ethnic group"
[2] "Ethnicity"
[3] "Gender"
[4] "Age"
[5] "Police District"
[6] "ANZSOC Division"
[7] "Seriousness of Offences"
[8] "Maximum Penalty for Offences (years)"
[9] "Method of Proceeding"

$type
[1] "rate" "prop" "rate" "rate" "rate" "prop" "prop" "prop" "prop"

$years
 [1] 2011 2012 2013 2014 2015 2016 2017 2018 2019 2020 2021
```

Now that we have a version of tableInfo() that works, we might also explore a vectorised version that does not need any loops. After exploring, we find that the vectorised version produces the same result as the version that is based on loops. The changes to vectorise the function involve almost the entire function. If we did not keep the old version, and wanted to revert to the old version, we would have to start again from scratch.

Manual Version Control

The manual version control approach involves simply copy and pasting our old code into another file and naming them based on the version information. Although, this only works in limited situations and if more changes are made, keeping track can get confusing.

```
NamedVersions
├── tableInfo-v1.R
├── tableInfo-v2.R
└── tableInfo-v3.R

0 directories, 3 files
```

```
NamedVersions
├── tableInfo-v1-skip-nrows.R
├── tableInfo-v2-by-type.R
└── tableInfo-v3-vec.R

0 directories, 3 files
```

```
FolderVersions
├── v1-skip-nrows
│   ├── tableEnd.R
│   ├── tableInfo.R
│   └── tableStart.R
├── v2-by-type
│   ├── tableEnd.R
│   ├── tableInfo.R
│   └── tableStart.R
└── v3-vec
    └── tableInfo.R

3 directories, 7 files
```

Git Version Control

We will work with the git2r package.

1. Import git2r
   ```
   library(git2r)
   ```
2. Initialise a repository (Creating a director and calling git2r::init())
   ```
   dir.create("tableInfo")
   repo <- init(repoDir)
   ```

2.1. If this is the first time you have used git2r, you need to provide configuration details
   ```
   config(repo, user.name="Brianna", user.email="Brianna@stat.auckland.ac.nz")
   ```

2.2. If a repository exists, you can open it in R using repository() function
   ```
   repo <- repository(repoDir)
   summary(repo)
   ```

3. Store functions, in separate files in the repository we just created, call status() to see untracked files
   ```
   status(repo)
   ```
   ```
   Untracked files:
       Untracked:   tableEnd.R
       Untracked:   tableInfo.R
       Untracked:   tableStart.R
   ```
4. When we want to add a file, we first add(), then commit()
   ```
   add(repo, c("tableEnd.R", "tableInfo.R", "tableStart.R"))
   commit(repo, "initial version")
   ```

Now after running status(repo), we can see a working directory clean message, and running summary(repo) will show information on the repository, such as our branches, commits, etc.

```
Local:     master /home/fos/SONAS/Files
Head:      [3026d64] 2024-05-07: initia

Branches:        1
Tags:            0
Commits:         1
Contributors:    1
Stashes:         0
Ignored files:   0
Untracked files: 0
Unstaged files:  0
Staged files:    0

Latest commits:
[3026d64] 2024-05-07: initial version
```

In our last example, we vectorised the function, meaning we no longer need the functions tableStart() and tableEnd(), therefore, we can remove these files from the repository.

```
rm_file(repo, c("tableStart.R", "tableEnd.R"))
status(repo)
```

```
Unstaged changes:
    Modified:   tableInfo.R

Staged changes:
    Deleted:    tableEnd.R
    Deleted:    tableStart.R
```

Once we have added and committed these changes, we have a repository with three different versions of our code (three "commits") recorded. In this add(), rather than specifying explicit files to add, we just say add all changes to the repository ("*").

```
add(repo, "*")
commit(repo, "Vectorised tableInfo(); removed tableStart() and tableEnd()")
status(repo)
working directory clean
```

Because we have a saved history, we can go back and view a previous version of a file

```
reflog(repo)
[abb4fca] HEAD@{0}: commit: Vectorised tableInfo(); removed tableStart() and tableEnd()
[5719574] HEAD@{1}: commit: Added 'by' and 'type' information
[3026d64] HEAD@{2}: commit (initial): initial version
```

```
viewCommit <- function(repo, commit, filename) {
  cat(content(git2r::tree(commits(repo)[[commit]])[filename]), sep="\n")
}
viewCommit(repo, 2, "tableInfo.R")
```

We can even view a file that used to exist, but has now been removed.

```
viewCommit(repo, 2, "tableStart.R")
```

It is also useful to view the files involved in a commit and to view the difference between two commits. The following functions provide a convenient interface for those tasks.

```
lsCommit <- function(repo, commit) {
   ls_tree(git2r::tree(commits(repo)[[commit]]))[c("path", "name")]
}

diffCommits <- function(repo, commit1, commit2) {
  commits <- commits(repo)
  cat(diff(git2r::tree(commits[[commit1]]),
        git2r::tree(commits[[commit2]]), as_char=TRUE))
}
```

Untracked
When you create a new file, Git doesn't know about it yet, it's untracked.

↓

Staged
When you add(), you tell Git: "I want to include this file in the next commit."

↓

Committed
Running commit() saves a snapshot of all staged changes. The commit is stored in the Git history with a unique ID

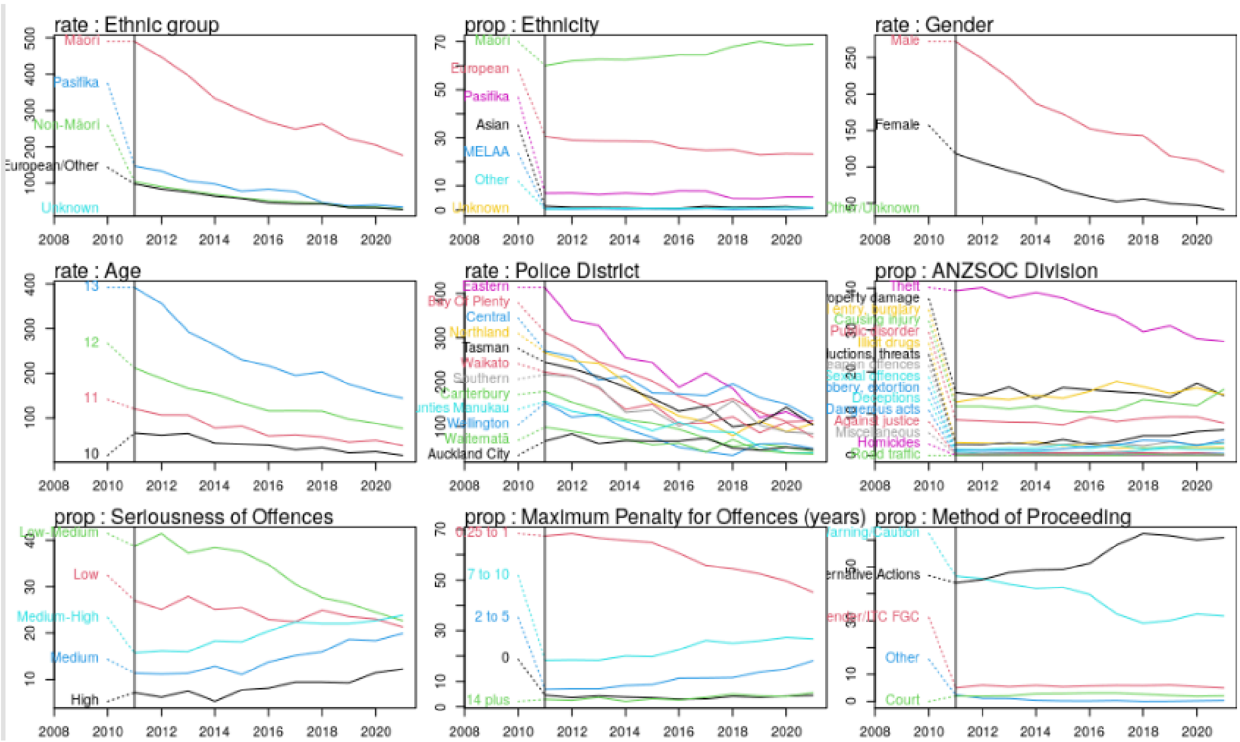
Code Development: Testing

The following function tableBy() uses the type information to decide whether to call getRateBy() or getPropBy().

```
csvFile <- "CSV/2021-1.1-Children.csv"

tableBy <- function(csv, skip, nrows, years, by, type) {
  if (type == "rate") {
    getRateBy(csv, skip, nrows,
              years, by)
  } else {
    getPropBy(csv, skip, nrows,
              years, by)
  }
}

info <- tableInfo(csvFile)
tables <- mapply(tableBy, info$skip, info$nrows, info$by, info$type,
                 MoreArgs=list(csv=csvFile, years=info$years),
                 SIMPLIFY=FALSE)
par(mfrow=n2mfrow(length(info$skip)), mar=c(2, 3, 2, 0))
invisible(mapply(plotTable, tables, info$by, info$type))
```



How do we check that our code is producing the correct result?

Checking for Weird

This approach acknowledges that we cannot check if the output is correct, but we can verify if the output is definitely incorrect. This may include basic summaries or plots to quickly check for things, such as negative counts. We can treat the output of our code as data and explore it for unusual or unexpected values.

```
summary(tables[[1]])
```

Test-Driven Development

Another approach is to create test-cases with known expected outputs. This can be achieved by using synthetic data which we manufacture the output, we will have a valid test case and expected output. For example, we have the tableStart() function that determines whether the ith value in the column is the start of the table. The inTable argument is a logical value telling us where the ith values are already within a table. We can generate our own sets of character, logical, and integer inputs for which we know the answer.

```
tableStart <- function(column, inTable, i) {
  !inTable && !is.na(column[i]) && !grepl("^Ratio", column[i]) && !is.na(column[i + 1])
}
```

These tests should return TRUE if the value at position i is the start of a table, and FALSE otherwise.

TEST 1: We're not already in a table	**TEST 2:** Current value column[i] is not missing
test1 <- rep("table", 2) [1] "table" "table" tableStart(test1, FALSE, 1) [1] TRUE tableStart(test1, TRUE, 1) [1] FALSE *The first value is fine, and next is also fine.*	test2 <- test1 test2[1] <- NA [1] NA "table" tableStart(test2, FALSE, 1) [1] FALSE tableStart(test2, TRUE, 1) [1] FALSE *Can't start a table with a missing value.*
TEST 3: The next value is not missing	**TEST 4:** The value is not something like "Ratio"
test3 <- test1 test3[2] <- NA [1] "table" NA tableStart(test3, FALSE, 1) [1] FALSE tableStart(test3, TRUE, 1) [1] FALSE *Can't start a table if the next value is missing.*	test4 <- test1 test4[1] <- "Ratio" [1] "Ratio" "table" tableStart(test4, FALSE, 1) [1] FALSE tableStart(test4, TRUE, 1) [1] FALSE *"Ratio" rows are ignored, not valid start.*

Now, we cannot say that our function is error-free because in order to make that claim, we would need to exhaust 100% of possible inputs, all we can deduce is that our function works for all our tests

Testing for Valid Input

As above, we have tested that our function produces the right output for the given inputs. Now we should test if the function handles incorrect inputs.

The tableStart() function assumes that inTable and i are both single values; what happens if we provide more than one value for those arguments?

Yes. It does, but we can modify our function so it provides a more helpful error message.

Before	After

`tableStart <- function(column, inTable, i) {`
`  !inTable &&`
`  !is.na(column[i]) &&`
`  !grepl("^Ratio", column[i]) &&`
`  !is.na(column[i + 1])`
`}`
	`tableStart <- function(column, inTable, i) {`
	`  if (!is.logical(inTable) || length(inTable) != 1 ||`
	`    !is.numeric(i) || length(i) != 1)`
	`    stop(paste("'inTable' should be logical;",`
	`          "'i' should be numeric;",`
	`          "both should be length 1"))`
	`  !inTable &&`
	`  !is.na(column[i]) &&`
	`  !grepl("^Ratio", column[i]) &&`
	`  !is.na(column[i + 1])`
	`}`

```
tableStart(test1, c(FALSE, TRUE), 1)
```

BEFORE: Error in !inTable && !is.na(column[i]): 'length = 2' in coercion to 'logical(1)'

AFTER: Error in tableStart(test1, c(FALSE, TRUE), 1): 'inTable' should be logical; 'i' should be numeric; both should be length 1

Regression Testing

When we refactored code into a function (or created a new version of a function), we have ensured the new function produces the same result as the code that it came from (or the previous version). This is known as regression testing.

Whenever we modify our code to make it better in some way, it is vital to check that we have not made our code worse in some other way.

If we obtain results for a data set that we believe are correct, we can save the correct results and then, whenever we modify our code, we can check that the code still produces the same correct results.

```
sameOutput <- sapply(tableIndex,
           function(i) {
             all.equal(tables[[i]],
                   read.csv(testTables[i], check.names=FALSE))
           })
sameOutput
[1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

Test Scripts

If a function is supposed to crash (error), we check that it does crash. We can write code that will generate an error if we do not get the correct result, we can do this using the stopifnot() function. This checks if a condition is TRUE. If it's not true, it stops the code and throws an error.

stopifnot(sameOutput)

We want the test to produce an error when there is an error, hence we use tools::assertError(). For example, the following code will stop (with an error) if the tableStart() call does not generate an error. tools::assertError() checks that a piece of code does generate an error.

tools::assertError(tableStart(test1, c(FALSE, TRUE), 1))

There is also a tools::assertWarning() function for checking that a piece of code will generate a warning.

Code Development: Debugging

1. Read the Error Message

Error in seq.default(yRange[2], yRange[1], length.out = length(order)): 'to' must be a finite number
The above error message suggests that it lies in the seq.default function, and 'to' must be a finite number suggests that the input could be NA, NaN or Inf.

2. Print the Call Stack

traceback()

```
5: mapply(plotTable, tablesYoung, infoYoung$by, infoYoung$type)
4: (function (data, by, yvar)
   {
       xRange <- range(data$year)
       yRange <- range(data[[yvar]], na.rm = TRUE)
       groups <- split(data, data[[by]])
       groupNames <- names(groups)
       starts <- sapply(groups, function(x) x[[yvar]][1])
       order <- order(starts, decreasing = TRUE)
       labelPos <- seq(yRange[2], yRange[1], length.out = length(order))
```

We see that the error was generated by the yRange[2], yRange[1], length.out = length(order) section which is stored in labelPos. labelPos is then called in the plot. The section is being called by seq(). Hence, we have clues as to something went wrong in seq() within plotTable()

3. Simplify the Problem

One of the things that we have learned from traceback() is that the problem occured within a call to plotTable(), but that in turn occurred within a call to mapply(). The mapply() function called plotTable() several times; our debugging task will be simpler if we can isolate which plotTable() call was causing the problem.

Because mapply(plotTable, ...) was used (calls plotTable() multiple times), you need to:

1. Switch to a for loop
2. Find out which table caused the crash

```
for (i in seq_along(tablesYoung)) {
  plotTable(tablesYoung[[i]], infoYoung$by[i], infoYoung$type[i])
}
```

This will print the iteration number before crashing. Now we know table 5 is the issue.

4. Add Print Statements

Edit plotTable() and add:

```
print(yRange)
print(order)
```

[1] "_" "984.76444887148273"

This means non-numeric data snuck in, like an underscore ("_").

5. Debug the Function

Using debug() on the plotTable()

```
debug(plotTable)
```

Now, when the function is called, you can:

1. Step line by line (n)
2. Inspect any variable just by typing its name (e.g. yRange, data[[yvar]])
3. Quit the browser with Q

This helps us see the problem live as it unfolds. Instead of manually debugging, you can tell R to trigger debugging automatically on error:

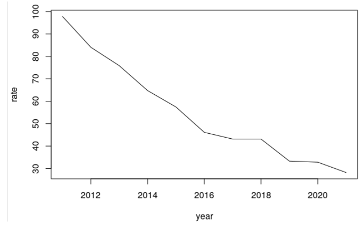
```
options(error = recover)
```

This will show a numbered list of the call stack and let you enter any frame to inspect variables when an error occurs.

Code Development: Graphics

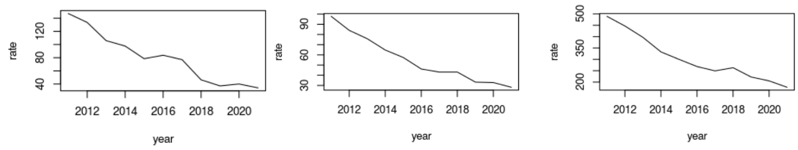
1. Splitting the childrenEthnicGroups DF by group and plotting the first group

```
childrenEthnicGroups <-
                split(childrenByEthnicGroup, childrenByEthnicGroup$group)
plot(rate ~ year, childrenEthnicGroups[[1]], type="l")
```



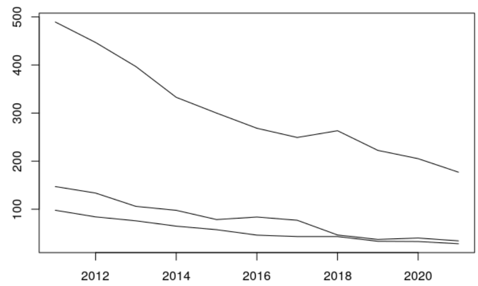
2. Running a loop to produce a plot for each group

```
par(mfrow=c(2, 2))
for (i in childrenEthnicGroups){
   plot(rate ~ year, i, type="l")
}
```



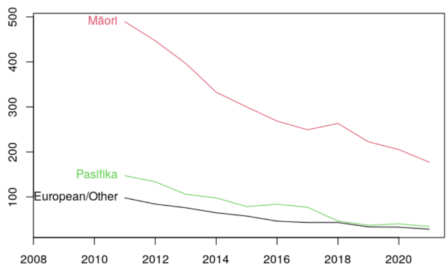
3. Use empty plot, iterate over DF to add line for each group as opposed to separate plots per group.

```
xRange <- range(childrenByEthnicGroup$year)
yRange <- range(childrenByEthnicGroup$rate)
plot.new()
plot.window(xRange, yRange)
box()
axis(1)
axis(2)
for (i in seq_along(childrenEthnicGroups)) {
   lines(rate ~ year, childrenEthnicGroups[[i]], type="l")
}
```



4. Add labels and color to plot so we can tell which line refers to which group

```
groupNames <- names(childrenEthnicGroups)
par(xpd=TRUE)
plot.new()
plot.window(c(min(xRange) - .25*diff(xRange), max(xRange)), yRange)
box()
axis(1)
axis(2)
for (i in seq_along(childrenEthnicGroups)) {
   lines(rate ~ year, childrenEthnicGroups[[i]], type="l", col=i)
   text(xRange[1], childrenEthnicGroups[[i]]$rate[1], groupNames[i],
       pos=2, adj=1, col=i)
}
```



Code Development: Consolidation-1

Refactoring getTable()

Problematic "_" values in the getTable() produced errors. With modular functions, it's easier to extract and refactor. The code below shows how we can view where the error starts. Instead of error-handling the plotTable() function itself, where the error resides, we can go lower-level to when we read the data in, to account for special cases such as "_" to read as NA.

```
getTable <- function(csvFile, skip, nrows, keep, yearNames, name, by) {
  cols <- c(1, keep)
  df <- read.csv(csvFile, skip=skip, nrows=nrows, header=FALSE, na.strings=c("NA", "_"))[cols]
  colnames(df) <- c(by, yearNames)
  dfLong <- reshape2::melt(df, id.vars=by, variable="year", value.name=name)
  dfLong$year <- as.numeric(as.character(dfLong$year))
  dfLong
}
```

Before	After

<pre>9   Wellington 2011 642.05276226392618
10   Tasman 2011 1059.2241145477387
11   Canterbury 2011 640.61154418762692
12   Southern 2011 858.89570552147256
13 Outside New Zealand (District) 2011   _
14   Northland 2012 830.5476910774189
15   Waitematā 2012 384.49800078185547</pre>
	<pre>9 Wellington 2011 642.0528
	10 Tasman 2011 1059.2241
	11 Canterbury 2011 640.6115
	12 Southern 2011 858.8957
	13 Outside New Zealand (District) 2011 NA
	14 Northland 2012 830.5477
	15 Waitematā 2012 384.4980</pre>

Regression Testing

Since we have altered the getTable() function and this function also has dependencies such as getRateBy(), getPropBy(), and getBy(), these functions are now at risk of being broken. We should test that the changes in getTable() have not broken anything.

```
tables <- mapply(tableBy, info$skip, info$nrows, info$by, info$type,
          MoreArgs=list(csv=csvFile, years=info$years),
          SIMPLIFY=FALSE)

testDir <- "YJI-2021"

tableIndex <- seq_along(tables)
testTables <- file.path(testDir, paste0("table-", tableIndex, ".csv"))

all(sapply(tableIndex,
       function(i) {all.equal(tables[[i]], read.csv(testTables[i], check.names=FALSE))
       }))
[1] TRUE
```

Code Development: Serialization

This topic focuses on storing R build data into external files. The following code takes the JYItables list and, for each original CSV file, writes out each individual table to its own CSV file.

```
outPath <- "Serialize"                  # Creates a folder path where the CSVs will be saved.
for (i in seq_along(YJItables)) {       # Loops over each sheet which has two elements: info and tables.
  sheet <- YJItables[[i]]               # Extracts one sheet, a list with two elements: info and tables.
  info <- sheet$info                    # Stores the metadata associated with this sheet
  for (j in seq_along(sheet$tables)) {      # Loops through each table in the current sheet.
    table <- sheet$tables[[j]]              # Extracts the actual data frame.
    tableName <- gsub(" ", "-", gsub("[()]", "", names(sheet$tables)[j]))
    write.csv(table,                    # writes the table as a .csv file
         file.path(outPath, paste0(names(YJItables)[i], "-", tableName, "-", info$type[j], ".csv")),
         row.names=FALSE)
  }
}
```

```
Serialize
├── 2018-children-Age-rate.csv
├── 2018-children-ANZSOC-Division-prop.csv
├── 2018-children-Ethnic-group-rate.csv
├── 2018-children-Ethnicity-prop.csv
├── 2018-children-Gender-prop.csv
├── 2018-children-Maximum-Penalty-for-Offences-years-prop.csv
├── 2018-children-Police-District-rate.csv
├── 2018-children-Seriousness-of-Offences-prop.csv
├── 2018-young-people-Age-rate.csv
├── 2018-young-people-ANZSOC-Division-prop.csv
├── 2018-young-people-Ethnic-group-rate.csv
├── 2018-young-people-Ethnicity-prop.csv
├── 2018-young-people-Gender-prop.csv
├── 2018-young-people-Maximum-Penalty-for-Offences-years-prop.csv
```

Code Development: Diffing R code

```
library("diffobj")
```

Comparing two functions	`diffobj::diffPrint(f1, f2)`
Comparing functions if R formats differently from original	`diffobj::diffFile("f1.R", "f2.R")`

Comparing two code chunks	`code1 <- quote(x <- 1)`
	`code2 <- quote(x <- 1:10)`
	`diffobj::diffPrint(code1, code2)`

Comparing two code chunks with multiple expressions	`code1 <- quote({`
	`x <- 1`
	`y <- 2 })`
	`code2 <- quote({`
	`x <- 1:10`
	`y <- 2:20 })`
	`diffobj::diffPrint(code1, code2)`

Code Development: Convert XLSX to CSV

This topic covers how to convert Excel Workbooks into CSV files as a code-based solution. There are five Excel workbooks to convert and we want to write out two sheets from each workbook.

```
Data
├── 08X5L0-Youth-Justice-Indicators-2019-August.xlsx
├── Youth-Justice-Indicators-2020-DECEMBER-FINAL-v2.0.xlsx
├── Youth-Justice-Indicators-2021-FINAL.xlsx
├── Youth-Justice-Indicators-APRIL2018-Workbook.xlsx
└── Youth-Justice-Indicators-April-2023-Workbook.xlsx

0 directories, 5 files
```

```
library(readxl)          # Loads in readxl package to read .xlsx files
outPath <- "Convert"     # Sets the output folder where the CSVs will be saved.

convertXLSX <- function(xlfile) {
  sheets <- excel_sheets(file.path(dataPath, xlfile)) # Lists all sheet names from dataPath/xlfile
  keepSheets <- grep("^YJI 1.1", sheets) # Filters sheets that start with "YJI 1.1", returning their index
  exportCSV <- function(sheet) {
    df <- read_xlsx(file.path(dataPath, xlfile), sheet=sheet)  # Reads Excel sheet into a DF
    yearStart <- regexpr("[0-9]{4}", xlfile) # Finds position of first 4-digit number in the filename
    year <- substring(xlfile, yearStart, yearStart + 3) # Extracts year from filename based on position
    safeSheet <- gsub(" ", "-", gsub("YJI |[()]", "", sheet))
    csvName <- file.path(outPath, paste0(year, "-", safeSheet, ".csv")) # Constructs full output path
    write.csv(df, csvName, row.names=FALSE) # Saves the df to CSV, excluding row numbers.
  }
  lapply(sheets[keepSheets], exportCSV) # Applies exportCSV() to each sheet in the Excel file
}
```

```
Convert
├── 2018-1.1-Age-10-13.csv
├── 2018-1.1-Age-14-16.csv
├── 2019-1.1-Age-10-13.csv
├── 2019-1.1-Age-14-16.csv
├── 2020-1.1-Children.csv
├── 2020-1.1-Young-people.csv
├── 2021-1.1-Children.csv
├── 2021-1.1-Young-people.csv
├── 2023-1.1-Children.csv
└── 2023-1.1-Young-people.csv

0 directories, 10 files
```

All Together

Introduction

We use R to verify the claims between the discrepancies between the media and official data of youth crime rates. Starting with a manual approach and explaining why this approach is flawed.

1. Open the Excel file, highlight the specific data, copy and paste this into a plain text file called manual-rate.txt. This data represents the offending rate per 10,000 people for three ethnic groups, over 11 years.
2. Reading the data in, read.table() reads data from a text file that is structured like a table
   ```
   rateManual <- read.table("Data/manual-rate.txt", header=TRUE)
   ```
3. Plotting the data, matplot() plots columns of a matrix or DF, each column becomes a separate line or set of points. t(rateManual) is used because rows represent ethnic groups, while columns represent years. matplot(), by default would plot each column as a separate series, plotting 11 lines, one for each year, with only 3 points per line. If we want to plot the ethnic group over time, we need to swap the rows and columns, hence, 3 lines (one for each ethnic group), and 11 points (for each year).
   ```
   matplot(t(rateManual), type="l")
   ```

The manual approach eliminates ant reproducibility, record, modification and is highly error-prone, hence we need to write R code that can automatically extract data.

Functions

We grab the offending rates by ethnic group by looking at the spreadsheet, we notice the data lives in rows 20-22 and columns 24-34. skip=19 tells R to completely ignore the first 19 rows of the CSV file. nrows=3 tells R that after skipping 19 rows of junk, only read the next 3 rows. [rateCols] is a filter. Of the data we just grabbed, only keep columns specified in rateCols (Columns 24 to 34). We then tidy the column names.

```
csvFile <- "CSV/2021-1.1-Children.csv"
rateCols <- 24:34
rateByEthnicGroup <- read.csv(csvFile, skip=19, nrows=3)[rateCols]
yearNames <- 2011:2021
colnames(rateByEthnicGroup) <- yearNames
```

The data does not include column names to tell us what different rows represent. We should modify the existing code.

```
rateByEthnicGroup <- read.csv(csvFile, skip=20, nrows=3,  header=FALSE)[c(1, rateCols)]
colnames(rateByEthnicGroup) <- c("group", yearNames)
```

The above modification skips the first 20 rows as opposed from the first 19, meaning we should set header=FALSE. The subsetting means keep the first column and the rateCols (24-34), effectively appending the ethnicity names to their rates. We label this as "group".

	group	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020	2021
1	Māori	489.39727	446.82714	396.68769	332.73327	299.96375	268.40380	249.22280	263.26996	222.3782	205.12684	176.99938
2	Pasifika	147.11407	133.55241	105.87843	97.66972	78.49106	83.73942	77.03974	46.41263	37.1390	40.08221	34.15912
3	European/Other	97.80214	84.04996	75.84956	64.74028	57.41285	46.11012	43.11791	43.11369	33.2715	32.79973	28.18008

DF is wide, we want a row to represent a unique observation. Use reshape2::melt(). See Figure 1.

```
rateByEthnicGroupLong <-reshape2::melt(rateByEthnicGroup, id.vars="group", variable="year", value.name="rate")
```

- id.vars="group" The group columns should stay as is (identifier)
- variable="year" Take all column names (2011, 2012, etc) and put them in new column "year"
- value.name="rate" Take all values from the columns and put them in a new column "rate"

Wide Format: 3 Rows, 12 Columns
Long Format: (Unique Identifier x Total Years) 33 Rows, 3 Columns

One problem with melting the DF is that the resulting year is a factor. R stores factors as characters, hence, converting straight to numeric would return internal integer codes, not the actual numbers.

```
class(rateByEthnicGroupLong$year)       # [1] "factor"
rateByEthnicGroupLong$year <- as.numeric(as.character(rateByEthnicGroupLong$year))
```

	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020	2021
1	489	446	396	332	399	269	249	263	222	205	176
2	147	133	105	97	78	83	77	46	37	40	34
3	97	84	75	64	57	46	43	43	33	32	28

⟶

group	year	rate
1	2011	489
2	2011	147
3	2011	97
1	2012	446
. . .		
3	2021	28

Figure 1: Reshaping the Offenders Rate from Wide to Long Format

With small modifications, we can write similar code to extract another table, the Count by Ethnicity. Now, if we compare the table extraction for the rate vs count, we see they are very similar.

```
rateByEthnicGroup <- read.csv(csvFile, skip=20, nrows=3, header=FALSE)[c(1, rateCol)]
countByEthnicGroup <- read.csv(csvFile, skip=20, nrows=3, header=FALSE)[c(1, countCols)]
colnames(rateByEthnicGroup) <- c("group", yearNames)
colnames(countByEthnicGroup) <- c("group", yearNames)
rateByEthnicGroupLong <-
  reshape2::melt(rateByEthnicGroup, id.vars="group", variable="year",  value.name="rate")
countByEthnicGroupLong <-
  reshape2::melt(countByEthnicGroup, id.vars="group", variable="year", value.name="count")
rateByEthnicGroupLong$year <- as.numeric(as.character(rateByEthnicGroupLong$year))
countByEthnicGroupLong$year <- as.numeric(as.character(countByEthnicGroupLong$year))
```

Therefore, this hints we should create a single function with parameter input.

```
tableByEthnicGroup <- function(keep, name){
  cols <- c(1, keep)
  df <- read.csv(csvFile, skip=20, nrows=3, header=FALSE)[cols]
  colnames(df) <- c("group", yearNames)
  dfLong <-  reshape2::melt(df, id.vars="group", variable="year",  value.name=value.name)
  dfLong$year <- as.numeric(as.character(dfLong$year))
}
```

We should check if the new function provides the same output as our singular functions.

```
all.equal(rateByEthnicGroupLong, tableByEthnicGroup(rateCols, "rate"))          # [1] TRUE
all.equal(countByEthnicGroupLong, tableByEthnicGroup(countCols, "count"))       # [1] TRUE
```

In Excel, there is a third table, population by ethnic group, this works for our tableByEthnicGroup function as well.

```
rateByEthnicGroupLong <- tableByEthnicGroup(rateCols, "rate")
countByEthnicGroupLong <- tableByEthnicGroup(countCols, "count")
popByEthnicGroupLong <- tableByEthnicGroup(popCols, "pop")
```

Since the data is in long format, we can actually combine all three, count, rate and population into one table. We could combine count and rate using merge, allowing two DF's to combine into a single DF. Or, we could use cbind() to combine more than two.

```
childrenByEthnicGroup <- merge(rateByEthnicGroupLong,
                               countByEthnicGroupLong, by=c("year", "group"))
childrenByEthnicGroup <- cbind(rateByEthnicGroupLong,
                               popByEthnicGroupLong[3], countByEthnicGroupLong[3] )
```

Further, there's granular breakdowns of ethnicities in another table in a different format. There's two tables, one for the number of offenders and one for the proportion per ethnicity, providing a promising pattern to work with.

```
getTable <- function(skip, nrows, keep, name, by){
  cols <- c(1, keep)
  df <- read.csv(csvFile, skip=skip, nrows=nrows, header=FALSE)[cols]
  colnames(df) <- c(by, yearNames)
  dfLong <-  reshape2::melt(df, id.vars=by, variable="year",  value.name=value.name)
  dfLong$year <- as.numeric(as.character(dfLong$year))
}
propByEthnicityLong <- getTable(61, 7, propCols, "prop", "ethnicity")
numberByEthnicityLong <- getTable(61, 7, numberCols, "number", "ethnicity")
childrenByEthnicity <- cbind(numberByEthnicityLong, propByEthnicityLong[3])
```

Looking further, we see more tables, this time by Gender. getTable() works for this dataset too!

```
rateByGenderLong <- getTable(75, 2, rateCols, "rate", "gender")
countByGenderLong <- getTable(75, 2, countCols, "count", "gender")
popByGenderLong <- getTable(75, 2, popCols, "pop", "gender")
childrenByGender <- cbind(countByGenderLong, popByGenderLong[3], rateByGenderLong[3])
```

childrenByEthnicity

ethnicity	year	number	prop
Māori	2011	2808	60.0128233
Pasifika	2011	326	6.9673007
Asian	2011	72	1.5387903
MELAA	2011	24	0.5129301
Other	2011	17	0.3633255
European	2011	1432	30.6048301

childrenByEthnicGroup

group	year	count	pop	rate
Māori	2011	2808	58369.97	489.39727
sifika	2011	326	22543.29	147.11407
/Other	2011	1545	160706.72	97.80214
Māori	2012	2605	59520.02	446.82714
sifika	2012	298	22780.30	133.55241
/Other	2012	1302	158149.70	84.04996

childrenByGender

	gender	year	count	pop	rate
1	Male	2011	3368	123990	271.63481
2	Female	2011	1391	117630	118.25215
3	Male	2012	3057	123250	248.03245
4	Female	2012	1235	117190	105.38442
5	Male	2013	2696	121750	221.43737
6	Female	2013	1099	116010	94.73321

Since we can now extract rates, counts and populations for ethnicities as well as for genders, we should combine them into a single function to eliminate more repeatability.

```
getRateBy <- function(skip, nrows, by) {
  rate <- getTable(skip, nrows, rateCols, "rate", by)
  count <- getTable(skip, nrows, countCols, "count", by)
  pop <- getTable(skip, nrows, popCols, "pop", by)
  cbind(count, pop[3], rate[3])
}
```

Offenders by Ethnic Group (20:23, A:AH)

```
getRateBy(20, 3, "group")
```

Ethnic group	Number of distinct offenders 2010/11	2011/12	2012/13	2013/14	2014/15	2015/16	2016/17	2017/18	2018/19	2019/20	2020/21
Māori	2,808	2,605	2,318	1,948	1,749	1,555	1,463	1,526	1,315	1,181	1,011
Pasifika	326	298	238	220	179	192	179	107	88	93	79
European/Other	1,545	1,302	1,144	951	830	661	625	613	476	451	376

Ethnic group	Population 2010/11	2011/12	2012/13	2013/14	2014/15	2015/16	2016/17	2017/18	2018/19	2019/20	2020/21
Māori	58,370	59,520	59,950	60,460	60,400	60,750	62,250	64,570	67,190	69,890	72,470
Pasifika	22,543	22,780	23,062	23,262	23,624	24,042	24,639	25,682	26,923	28,166	29,343
European/Other	160,707	158,150	154,738	151,699	149,756	150,318	153,711	158,388	162,557	166,914	169,287

Ethnic group	Rate per 10,000 Population (adjusted)* 2010/11	2011/12	2012/13	2013/14	2014/15	2015/16	2016/17	2017/18	2018/19	2019/20	2020/21
Māori	489	447	397	333	300	268	249	263	222	205	177
Pasifika	147	134	106	98	78	84	77	46	37	40	34
European/Other	98	84	76	65	57	46	43	43	33	33	28

Offenders by Gender (75:77, A:AH)

```
getRateBy(75, 2, "gender")
```

Gender	Number of distinct offenders 2010/11	2011/12	2012/13	2013/14	2014/15	2015/16	2016/17	2017/18	2018/19	2019/20	2020/21
Male	3,368	3,057	2,696	2,249	2,066	1,839	1,793	1,822	1,514	1,484	1,305
Female	1,391	1,235	1,099	970	790	684	610	678	619	609	550

Gender	Population 2010/11	2011/12	2012/13	2013/14	2014/15	2015/16	2016/17	2017/18	2018/19	2019/20	2020/21
Male	123,990	123,250	121,750	120,500	119,640	120,630	123,330	127,470	131,790	136,260	139,470
Female	117,630	117,190	116,010	114,920	114,140	114,480	117,270	121,180	124,880	128,720	131,630

Gender	Rate per 10,000 Population 2010/11	2011/12	2012/13	2013/14	2014/15	2015/16	2016/17	2017/18	2018/19	2019/20	2020/21
Male	272	248	221	187	173	152	145	143	115	109	94
Female	118	105	95	84	69	60	52	56	50	47	42

Offenders by Age (85:89, A:AH)

```
getRateBy(85, 4, "age")
```

Age	Number of distinct offenders 2010/11	2011/12	2012/13	2013/14	2014/15	2015/16	2016/17	2017/18	2018/19	2019/20	2020/21
10	404	362	371	251	242	233	176	219	145	164	111
11	723	655	627	441	476	353	373	358	303	337	259
12	1,263	1,140	1,019	910	766	679	691	697	612	588	518
13	2,370	2,136	1,779	1,619	1,373	1,260	1,164	1,228	1,075	1,005	972

Age	Population 2010/11	2011/12	2012/13	2013/14	2014/15	2015/16	2016/17	2017/18	2018/19	2019/20	2020/21
10	61,070	58,790	56,830	57,770	58,370	59,130	61,460	65,410	66,560	67,130	68,300
11	60,330	61,330	58,850	57,030	58,280	58,960	59,850	62,220	65,990	67,380	67,620
12	59,720	60,410	61,440	59,060	57,530	58,820	59,710	60,560	62,900	66,790	67,910
13	60,500	59,920	60,630	61,560	59,600	58,200	59,580	60,450	61,220	63,670	67,270

Age	Rate per 10,000 Population 2010/11	2011/12	2012/13	2013/14	2014/15	2015/16	2016/17	2017/18	2018/19	2019/20	2020/21
10	66	62	65	43	41	39	29	33	22	24	16
11	120	107	107	77	82	60	62	58	46	50	38
12	211	189	166	154	133	115	116	115	97	88	76
13	392	356	293	263	230	216	195	203	176	158	144

Since we can now extract the number and proportion of offenders for ethnicities, we should combine them into a single function to eliminate more repeatability.

```
getPropBy <- function(skip, nrows, by) {
  number <- getTable(skip, nrows, numberCols, "number", by)
  prop <- getTable(skip, nrows, propCols, "prop", by)
  cbind(number, prop[3])
}
```

Offenders by Ethnicity (61:67, A:W)

```
getPropBy(61, 7, "ethnicity")
```

Ethnicity	2010/11	2011/12	2012/13	2013/14	2014/15	2015/16	2016/17	2017/18	2018/19	2019/20	2020/21
Māori	2,808	2,605	2,318	1,948	1,749	1,555	1,463	1,526	1,315	1,181	1,011
Pasifika	326	298	238	220	179	192	179	107	88	93	79
Asian	72	44	38	29	17	16	33	24	22	24	10
MELAA	24	16	18	10	14	8	14	5	8	5	10
Other	17	24	28	22	17	15	15	20	15	18	15
European	1,432	1,218	1,060	890	782	622	563	564	431	404	341
Unknown	81	88	96	102	99	117	137	256	256	369	394

Ethnicity	2010/11	2011/12	2012/13	2013/14	2014/15	2015/16	2016/17	2017/18	2018/19	2019/20	2020/21
Māori	60	62	63	62	63	65	65	68	70	68	69
Pasifika	7	7	6	7	6	8	8	5	5	5	5
Asian	2	1	1	1	1	1	1	1	1	1	1
MELAA	1	0	0	0	1	0	1	0	0	0	1
Other	0	1	1	1	1	1	1	1	1	1	1
European	31	29	29	29	28	26	25	25	23	23	23
Unknown											

Style - Checking Assumptions

Based on the getTable() function, we assume that the first three inputs are numbers and the last two are character values. We can check these by adding more code.

```
getTable <- function(skip, nrows, keep, name, by) {
  if (!(is.numeric(skip) && is.numeric(nrows) && is.numeric(keep) &&
      is.character(name) && is.character(by)))
    stop("'skip', 'nrows', and 'keep' must be numeric and 'name' and 'by' must be character values")
  ...
}
```

Let's say this input has been provided: `rateByEthnicGroup <- getTable(20, 3, 24, "rate", "group")`

```
getTable <- function(20, 3, 24, "rate", "group"){
   cols <- c(1, keep)        # [1] 1 24
   df <- read.csv(csvFile, skip=20, nrows=3, header=FALSE)[1, 24]   df =
   colnames(df) <- c(by, yearNames)       # [1] "group" "2011" "2012" ,..., "2021"
   Error in names(x) <- value: 'names' attribute [12] must be the same length as the vector [2]
```

```
            V1       V24
1        Māori 489.39727
2     Pasifika 147.11407
3 European/Other  97.80214
```

getTable() assumes the 'keep' input is the same length as yearNames. Since 'keep' = 24, two columns get extracted. When renaming columns, getTable() tries to name columns V1 and V2 using this list: ["group", "2011", "2012", "2013", "2014", "2015", "2016", "2017", "2018", "2019", "2020", "2021"]

Therefore, assumption checking can be implemented.

```
if (length(keep) != length(yearNames))
     stop("'keep' and 'yearNames' must have the same length")
```

Refactoring

We have refactored the getTable() function so csvFile and yearNames are no longer Global Variables.

```
getTable <- function(csvFile, skip, nrows, keep, yearNames, name, by){
   cols <- c(1, keep)
   df <- read.csv(csvFile, skip=skip, nrows=nrows, header=FALSE)[cols]
   colnames(df) <- c(by, yearNames)
   dfLong <-  reshape2::melt(df, id.vars=by, variable="year",  value.name=value.name)
   dfLong$year <- as.numeric(as.character(dfLong$year))
   dfLong
}
```

We have refactored the getRateBy() to eliminate Global Variables

```
getRateBy <- function(csv, skip, nrows, years, by) {
   nYears <- length(years)
   countCols <- 1:nYears + 1
   popCols <- countCols + nYears
   rateCols <- popCols + nYears
   rate <- getTable(csv, skip, nrows, rateCols, years, "rate", by)
   count <- getTable(csv, skip, nrows, countCols, years, "count", by)
   pop <- getTable(csv, skip, nrows, popCols, years, "pop", by)
   cbind(count, pop[3], rate[3])
}
```

We have refactored the getPropBy() to eliminate Global Variables

```
getPropBy <- function(csv, skip, nrows, years, by) {
   nYears <- length(years)
   numberCols <- 1:nYears + 1
   propCols <- numberCols + nYears
   number <- getTable(csv, skip, nrows, numberCols, years, "number", by)
   prop <- getTable(csv, skip, nrows, propCols, years, "prop", by)
   cbind(number, prop[3])
}
```

Divide and Conquer

getRateBy() and getPropBy() are refactored, although, we can still simplify by reducing the number of parameter inputs required. Looking at the data, we see that a table starts when there are two non-NA values consecutively.

1. Write a simple loop to check for two consecutive non-NA values in the dataset.

```
csvFile <- "CSV/2021-1.1-Children.csv"
firstColumn <- read.csv(csvFile, header=FALSE)[[1]]
for (i in 1:21) {
  if (!is.na(firstColumn[i])) {
    if (!is.na(firstColumn[i + 1])) {
      cat("table starts on row", i, "\n")
    }}}
```

```
[15] NA
[16] NA
[17] "Number of distinct offenders, and rate per 1
[18] NA
[19] "Ethnic group"
[20] NA
[21] "Māori"
[22] "Pasifika"
[23] "European/Other"
[24] "Unknown"
[25] "Non-Māori"
[26] "Total"
[27] "Ratio (Māori to Pasifika)"
[28] "Ratio (Māori to European/Other)"
[29] "Ratio (Māori to Non-Māori)"
[30] "*Considerable growth in the extent to which
```

2. If "Ratio" is in the row, treat this like an NA value and determine if we are in a table already.
This code checks if we aren't already in a table, the row we are at is non-NA, does not contain 'Ratio' and the following row is also non-NA, we have found the start of a table and we are now in a table. If we are in a table and the row we are on says "Total", then we have reached the end of the table.

```
inTable <- FALSE
for (i in 1:30) {
  if (!inTable && !is.na(firstColumn[i]) && !grepl("Ratio", firstColumn[i])) {
    if (!is.na(firstColumn[i + 1])) {
      cat("table starts on row", i, "\n")  # table starts on row 21
      inTable <- TRUE
    } }
  if (inTable && firstColumn[i] == "Total") {
    cat("table ends on row", i, "\n")      # table ends on row 26
  }}
```

3. Now that the code is increasing in complexity, we should use modular functions to break down.

```
inTable <- FALSE
for (i in 1:30) {
  if (tableStart(firstColumn, inTable, i)) {
    cat("table starts on row", i, "\n")
    inTable <- TRUE
  }  # table starts on row 21
  if (tableEnd(firstColumn, inTable, i)) {
    cat("table ends on row", i, "\n")
  } # table ends on row 26
}
```

```
tableEnd <- function(column, inTable, i) {
  inTable && column[i] == "Total"
}
```

```
tableStart <- function(column, inTable, i) {
  !inTable &&
  !is.na(column[i]) &&
  !grepl("^Ratio", column[i]) &&
  !is.na(column[i + 1])
}
```

4. Now working our way back to our getTable() function, we need to make some changes.
   1. Rather than the row the table starts on, we need the number of rows of skip (i-1)
   2. Rather than row of table end, we need the number of rows the table has (i - tableSkip - 1)

```
tableInfo <- function(csvFile) {
  csv <- read.csv(csvFile, header=FALSE)
  column <- csv[,1]
  inTable <- FALSE
  tableSkip <- numeric()
  tableRows <- numeric()
  numTables <- 0
  for (i in 1:length(column)) {
    if (tableStart(column, inTable, i)) {
      numTables <- numTables + 1
      tableSkip[numTables] <- i - 1
      inTable <- TRUE
    }
    if (tableEnd(column, inTable, i)) {
      tableRows[numTables] <- i - tableSkip[numTables] - 1
      inTable <- FALSE
    }
  }
  list(skip=tableSkip, nrows=tableRows)
}
tableInfo(csvFile)
$skip
[1]  20  61  75  85  96 141 164 176 188 196
$nrows
[1]  5  7  3  4 12 16  5  5  5
```

tableInfo() Explanation

First, we read the CSV file then store all the rows for the first column in 'column'. inTable is initialised to FALSE as we haven't started traversing yet. We initialise two numeric vectors, tableSkip and tableRows and counter numTables to 0. Then we enter the loop to check every row in the first column of the CSV. Traversing each row, if we are at the start of a table, increment the numTables count, append the row number - 1 to tableSkip to log how many rows it took to arrive at a table, then change inTable to TRUE, because now we are in a table. Then we check if the table has ended. If it has, then append to tableRows the number of rows the table has. This is calculated by the row we are on subtracted by all the rows we skipped to arrive at the start of the table, subtracted by 1 because the end of a table is not a data entry. Since we're no longer in a table, set inTable to FALSE. We then return a list of two vectors. The first vector contains the numbers of rows to skip for each table found, while the second vector contains their corresponding number of rows.

```
childrenByEthnicGroup <-  getRateBy(csvFile, info$skip[1], info$nrows[1], yearNames, "group")
childrenByEthnicity <- getPropBy(csvFile, info$skip[2], info$nrows[2], yearNames, "ethnicity")
```

Version Control

In this chapter we will automate three things. First, stop explicitly calling getRateBy() or getPropBy() based on whether the table contains rates,counts and populations or proportions and numbers. Then stop explicitly providing the by label, and finally, stop explicitly specifying yearNames. We can do this by modifying the tableInfo() function to return by, type and years, along with skip and nrows.

Now say we have modified tableInfo() but want to experiment with a vectorised approach rather than loops. Say we have successfully created a vectorised modification too. Now, both the loop and vectorised version of tableInfo() are very different from each other, we need to control the version.

```
library(git2r)
repoDir <- "tableInfo"
dir.create(repoDir)
repo <- init(repoDir)
config(repo, user.name="Paul", user.email="paul@stats.nz") # Or repo <- repository(repoDir)
status(repo)
```

```
Untracked files:
    Untracked:   tableEnd.R
    Untracked:   tableInfo.R
    Untracked:   tableStart.R
```

```
add(repo, c("tableEnd.R", "tableInfo.R", "tableStart.R"))
commit(repo, "initial version")
status(repo)
```

```
working directory clean
```

```
summary(repo)         ---------------------------------------->
```

```
Local:    master /home/fos/SONAS/Files/
Head:     [3026d64] 2024-05-07: initial

Branches:         1
Tags:             0
Commits:          1
Contributors:     1
Stashes:          0
Ignored files:    0
Untracked files:  0
Unstaged files:   0
Staged files:     0

Latest commits:
[3026d64] 2024-05-07: initial version
```

```
cat(diff(repo, as_char=TRUE))     # View the differences
reflog(repo)                      # View commit history -->
```

```
[5719574] HEAD@{0}: commit: Added 'by' and 'type' information
[3026d64] HEAD@{1}: commit (initial): initial version
```

```
rm_file(repo, c("tableStart.R", "tableEnd.R"))     # Remove Files in Repository

viewCommit <- function(repo, commit, filename) {# Function to view commits (including deleted file)
   cat(content(git2r::tree(commits(repo)[[commit]])[filename]), sep="\n")
}
lsCommit <- function(repo, commit) {
   ls_tree(git2r::tree(commits(repo)[[commit]]))[c("path", "name")]
} # Find what files existed in project at the time of a specific historical commit

diffCommits <- function(repo, commit1, commit2) {
  commits <- commits(repo)
  cat(diff(git2r::tree(commits[[commit1]]),
        git2r::tree(commits[[commit2]]), as_char=TRUE))
} # Find the exact line-by-line changes between two different historical commits
```