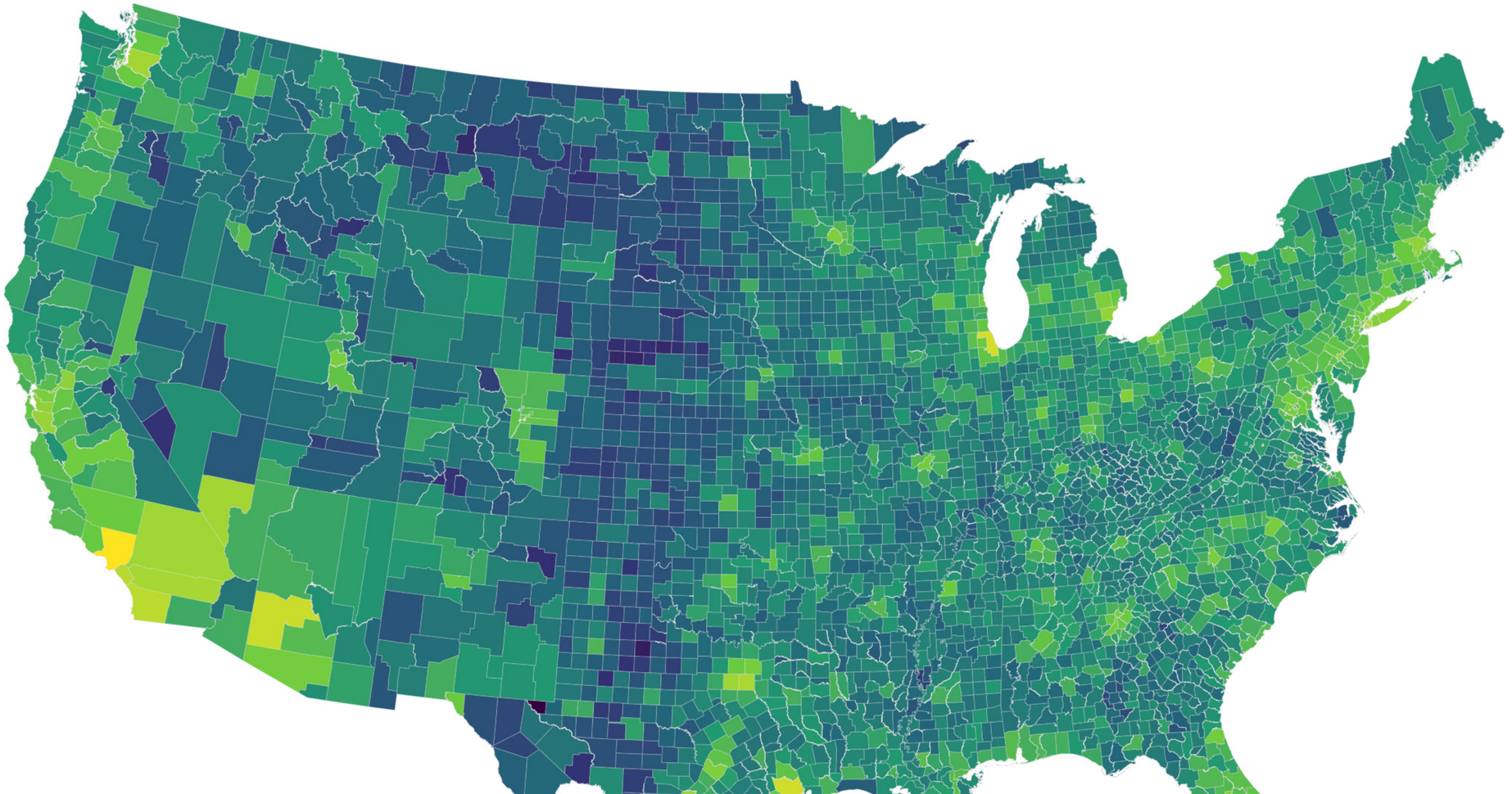
```r
nummodeguard <- function(x) {
  if (!is.vector(x)) {
    stop("Input x must be a vector.")
  }

  freq <- table(x)
  as.numeric(names(freq)[which.max(freq)])
}
```

Last time

- **Cartography** and **GIS**



This time

- Wrapping up odds and ends
  - Statistics in R
    - Hypothesis tests
    - Models and model plots
  - Flow control
    - Loops
    - Functions

Statistics

Specifically, statistical inference

- There are a variety of different kinds of statistical tests you may want to run
    - There are also a variety of things you may want to do *after* statistical tests, visually or otherwise
- Most standard tests have routines in base R. The more exotic you get, though, the more likely you'll need to turn to supplemental library

	Categorical	Continuous
Categorical	Tabular Analysis	Probit/Logit
Continuous	Difference of Means	Correlation Coefficient; Regression

“Simple” Hypothesis Tests

- Tabular analysis w/ chi squared
- Difference in means tests
- Correlation coefficients

- With these methods, we are typically looking for a single (or even no) estimate, and then a critical value and p-value

Tabular analysis w/ chi square

- Let's use our old friend, nes2000subset
  - `table(var1, var2)`
  - `chisq.test(var1, var2)`
- Simple

```
> nes2000<-read.csv("nes2000subset.csv")
> table(nes2000$partisanship, nes2000$vote)

    1   2
0 257  10
1 171  25
2 121  30
3  34  37
4  25 111
5  18 118
6   7 161
> chisq.test(nes2000$partisanship, nes2000$vote)

   Pearson's Chi-squared test

data:  nes2000$partisanship and nes2000$vote
X-squared = 653.9, df = 6, p-value < 2.2e-16
```

Difference in means t-test

- Genearlly, two different ways you might wish to run a difference in means t.test:
  - To test whether two groups are equal on some variable
    - *Does the amount of money someone makes vary by whether they vote democratic or republican?*
  - To test whether to separate variables are equal
    - *Do national parks have the same number of visitors in year 1 as opposed to year 2?*

Difference in means t-test: groups on one variable

- The “formula” specification
  - `ttest(variable~group, paired=F, conf.level=95)`
- Also simple!

```
> t.test(income~vote, data=nes2000, conf.level=.95)

    Welch Two Sample t-test

data:  income by vote
t = -3.978, df = 990, p-value = 7.457e-05
alternative hypothesis: true difference in means bet
to 0
95 percent confidence interval:
 -1.3583340 -0.4608969
sample estimates:
mean in group 1 mean in group 2
       6.592417        7.502033
```

Difference in means t-test: two variables on same units

- Let's use our old friend, nes2000subset
- The "two vectors" specification
  - `ttest(group1, group2, paired=F, conf.level=95)`
- Also simple!

```
> nps<-read.csv("npsvisitation0910.csv")
> t.test(nps$year2009, nps$year2010, paired=F, con

    Welch Two Sample t-test

data:  nps$year2009 and nps$year2010
t = 0.095024, df = 721.16, p-value = 0.9243
alternative hypothesis: true difference in means i
95 percent confidence interval:
 -232235.5  255860.0
sample estimates:
mean of x mean of y
 788894.5  777082.2
```

Correlation coefficients

- cor.test(). Two specifications:
  - By formula:
    - `cor.test(~partisanship+income, data=nes2000, method="pearson", conf.level=.95)`
  - By vectors:
    - `cor.test(nes2000$partisanship, nes2000$income, method="pearson", conf.level=.95)`
- Same result! Still easy!

```
> cor.test(~partisanship+income, data=nes2000, method="pear

    Pearson's product-moment correlation

data:  partisanship and income
t = 6.4724, df = 1123, p-value = 1.438e-10
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.1326622 0.2453647
sample estimates:
      cor
0.1896381

> cor.test(nes2000$partisanship, nes2000$income, method="pe

    Pearson's product-moment correlation

data:  nes2000$partisanship and nes2000$income
t = 6.4724, df = 1123, p-value = 1.438e-10
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.1326622 0.2453647
sample estimates:
      cor
0.1896381
```

Working with models

- Specifically:
  - OLS Regression (*the linear model*)
  - Probit, Logit, and other MLE-based methods (*the generalized linear model)*
- With these, we have the chance of multiple right-side variables
  - And thus, multiple parameters/statistics we are estimating

Bivariate Regression

```
> lm(partisanship~income, data=nes2000)

Call:
lm(formula = partisanship ~ income, data = nes2000)

Coefficients:
(Intercept)       income
     1.8489       0.1097
```

- How . . . underwhelming
  - Need to save result as an object
  - And then manipulate it

Bivariate Regression

```
> lm(partisanship~income, data=nes2000)

Call:
lm(formula = partisanship ~ income, data = nes2000)

Coefficients:
(Intercept)       income
     1.8489       0.1097
```

- How . . . underwhelming
  - Need to save result as an object
  - And then manipulate it

```
> model<-lm(partisanship~income, data=nes2000)
> summary(model)

Call:
lm(formula = partisanship ~ income, data = nes2000)

Residuals:
    Min      1Q  Median      3Q     Max
-4.1535 -1.9586 -0.3976  1.9439  4.0414

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.84886    0.13469  13.727  < 2e-16 ***
income       0.10975    0.01696   6.472 1.44e-10 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.146 on 1123 degrees of freedom
Multiple R-squared:  0.03596,  Adjusted R-squared:  0.0351
F-statistic: 41.89 on 1 and 1123 DF,  p-value: 1.438e-10
```

Bivariate Regression

```
> model<-lm(partisanship~income, data=nes2000)
> summary(model)

Call:
lm(formula = partisanship ~ income, data = nes2000)

Residuals:
    Min      1Q  Median      3Q     Max
-4.1535 -1.9586 -0.3976  1.9439  4.0414

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.84886    0.13469  13.727  < 2e-16 ***
income       0.10975    0.01696   6.472 1.44e-10 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.146 on 1123 degrees of freedom
Multiple R-squared:  0.03596,  Adjusted R-squared:  0.0351
F-statistic: 41.89 on 1 and 1123 DF,  p-value: 1.438e-10
```

- Lots of good stuff here!
  - **Intercept term**

Bivariate Regression

```
> model<-lm(partisanship~income, data=nes2000)
> summary(model)

Call:
lm(formula = partisanship ~ income, data = nes2000)

Residuals:
    Min      1Q  Median      3Q     Max
-4.1535 -1.9586 -0.3976  1.9439  4.0414

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.84886    0.13469  13.727  < 2e-16 ***
income       0.10975    0.01696   6.472 1.44e-10 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.146 on 1123 degrees of freedom
Multiple R-squared:  0.03596,  Adjusted R-squared:  0.0351
F-statistic: 41.89 on 1 and 1123 DF,  p-value: 1.438e-10
```

- Lots of good stuff here!
  - Intercept term
  - **Slope term (b)**

Bivariate Regression

```
> model<-lm(partisanship~income, data=nes2000)
> summary(model)

Call:
lm(formula = partisanship ~ income, data = nes2000)

Residuals:
    Min      1Q  Median      3Q     Max
-4.1535 -1.9586 -0.3976  1.9439  4.0414

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.84886    0.13469  13.727  < 2e-16 ***
income       0.10975    0.01696   6.472 1.44e-10 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.146 on 1123 degrees of freedom
Multiple R-squared:  0.03596,  Adjusted R-squared:  0.0351
F-statistic: 41.89 on 1 and 1123 DF,  p-value: 1.438e-10
```

- Lots of good stuff here!
  - Intercept term
  - Slope term
  - **Standard errors**

Bivariate Regression

```
> model<-lm(partisanship~income, data=nes2000)
> summary(model)

Call:
lm(formula = partisanship ~ income, data = nes2000)

Residuals:
    Min      1Q  Median      3Q     Max
-4.1535 -1.9586 -0.3976  1.9439  4.0414

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.84886    0.13469  13.727  < 2e-16 ***
income       0.10975    0.01696   6.472 1.44e-10 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.146 on 1123 degrees of freedom
Multiple R-squared:  0.03596,  Adjusted R-squared:  0.0351
F-statistic: 41.89 on 1 and 1123 DF,  p-value: 1.438e-10
```

- Lots of good stuff here!
  - Intercept term
  - Slope term
  - Standard errors
  - **T-values**

Bivariate Regression

```
> model<-lm(partisanship~income, data=nes2000)
> summary(model)

Call:
lm(formula = partisanship ~ income, data = nes2000)

Residuals:
    Min      1Q  Median      3Q     Max
-4.1535 -1.9586 -0.3976  1.9439  4.0414

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.84886    0.13469  13.727  < 2e-16 ***
income       0.10975    0.01696   6.472 1.44e-10 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.146 on 1123 degrees of freedom
Multiple R-squared:  0.03596,  Adjusted R-squared:  0.0351
F-statistic: 41.89 on 1 and 1123 DF,  p-value: 1.438e-10
```

- Lots of good stuff here!
  - Intercept term
  - Slope term
  - Standard errors
  - T-values
  - **P-values**

Bivariate Regression

```
> model<-lm(partisanship~income, data=nes2000)
> summary(model)

Call:
lm(formula = partisanship ~ income, data = nes2000)

Residuals:
    Min      1Q  Median      3Q     Max
-4.1535 -1.9586 -0.3976  1.9439  4.0414

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.84886    0.13469  13.727  < 2e-16 ***
income       0.10975    0.01696   6.472 1.44e-10 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.146 on 1123 degrees of freedom
Multiple R-squared:  0.03596,  Adjusted R-squared:  0.0351
F-statistic: 41.89 on 1 and 1123 DF,  p-value: 1.438e-10
```

- Lots of good stuff here!
  - Intercept term
  - Slope term
  - Standard errors
  - T-values
  - P-values
  - **"Stars"**

Bivariate Regression

```
> model<-lm(partisanship~income, data=nes2000)
> summary(model)

Call:
lm(formula = partisanship ~ income, data = nes2000)

Residuals:
    Min      1Q  Median      3Q     Max
-4.1535 -1.9586 -0.3976  1.9439  4.0414

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.84886    0.13469  13.727  < 2e-16 ***
income       0.10975    0.01696   6.472 1.44e-10 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.146 on 1123 degrees of freedom
Multiple R-squared:  0.03596,  Adjusted R-squared:  0.0351
F-statistic: 41.89 on 1 and 1123 DF,  p-value: 1.438e-10
```

- Lots of good stuff here!
  - Intercept term
  - Slope term
  - Standard errors
  - T-values
  - P-values
  - “Stars”
  - **R^2 (of various kinds)**

Bivariate Regression

```
> model<-lm(partisanship~income, data=nes2000)
> summary(model)

Call:
lm(formula = partisanship ~ income, data = nes2000)

Residuals:
    Min      1Q  Median      3Q     Max
-4.1535 -1.9586 -0.3976  1.9439  4.0414

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.84886    0.13469  13.727  < 2e-16 ***
income       0.10975    0.01696   6.472 1.44e-10 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.146 on 1123 degrees of freedom
Multiple R-squared:  0.03596,  Adjusted R-squared:  0.0351
F-statistic: 41.89 on 1 and 1123 DF,  p-value: 1.438e-10
```

- Lots of good stuff here!
  - Intercept term
  - Slope term
  - Standard errors
  - T-values
  - P-values
  - “Stars”
  - R2 (of various kinds)
  - **F-statistic**

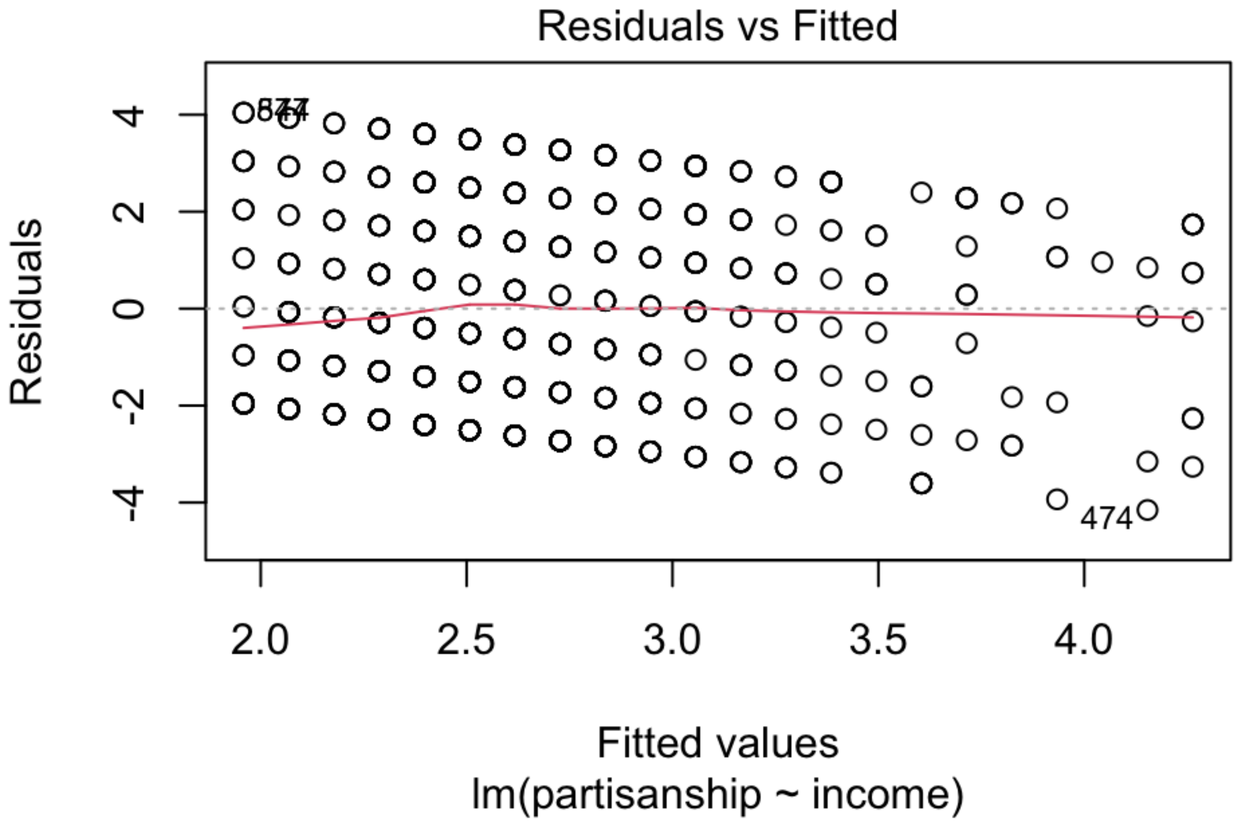
Bivariate Regression

```
> model<-lm(partisanship~income, data=nes2000)
> summary(model)

Call:
lm(formula = partisanship ~ income, data = nes2000)

Residuals:
    Min      1Q  Median      3Q     Max
-4.1535 -1.9586 -0.3976  1.9439  4.0414

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.84886    0.13469  13.727  < 2e-16 ***
income       0.10975    0.01696   6.472 1.44e-10 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.146 on 1123 degrees of freedom
Multiple R-squared:  0.03596,  Adjusted R-squared:  0.0351
F-statistic: 41.89 on 1 and 1123 DF,  p-value: 1.438e-10
```

- Lots of good stuff here!
  - Intercept term
  - Slope term
  - Standard errors
  - T-values
  - P-values
  - “Stars”
  - R2 (of various kinds)
  - F-statistic
  - **Residuals**

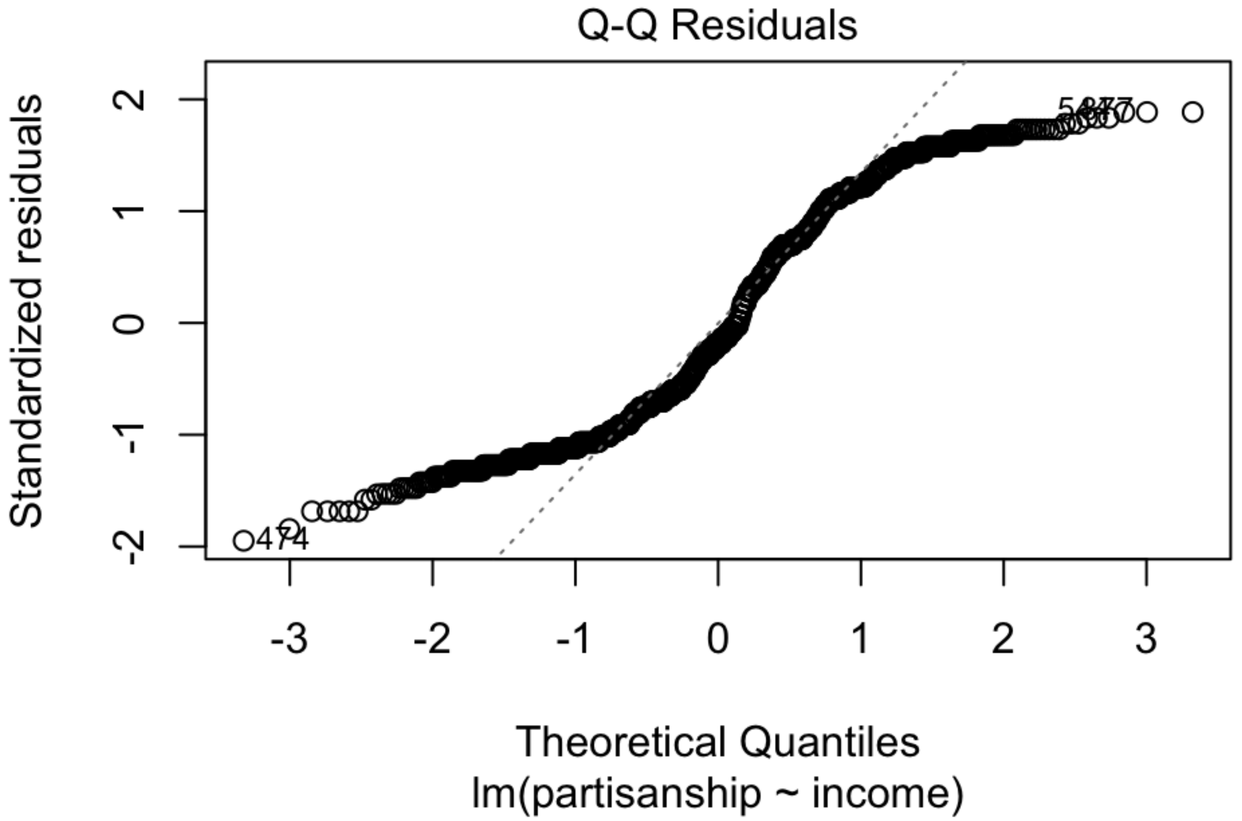
Diagnostics

- Basic diagnostics are very easy!
  - `plot(model)`



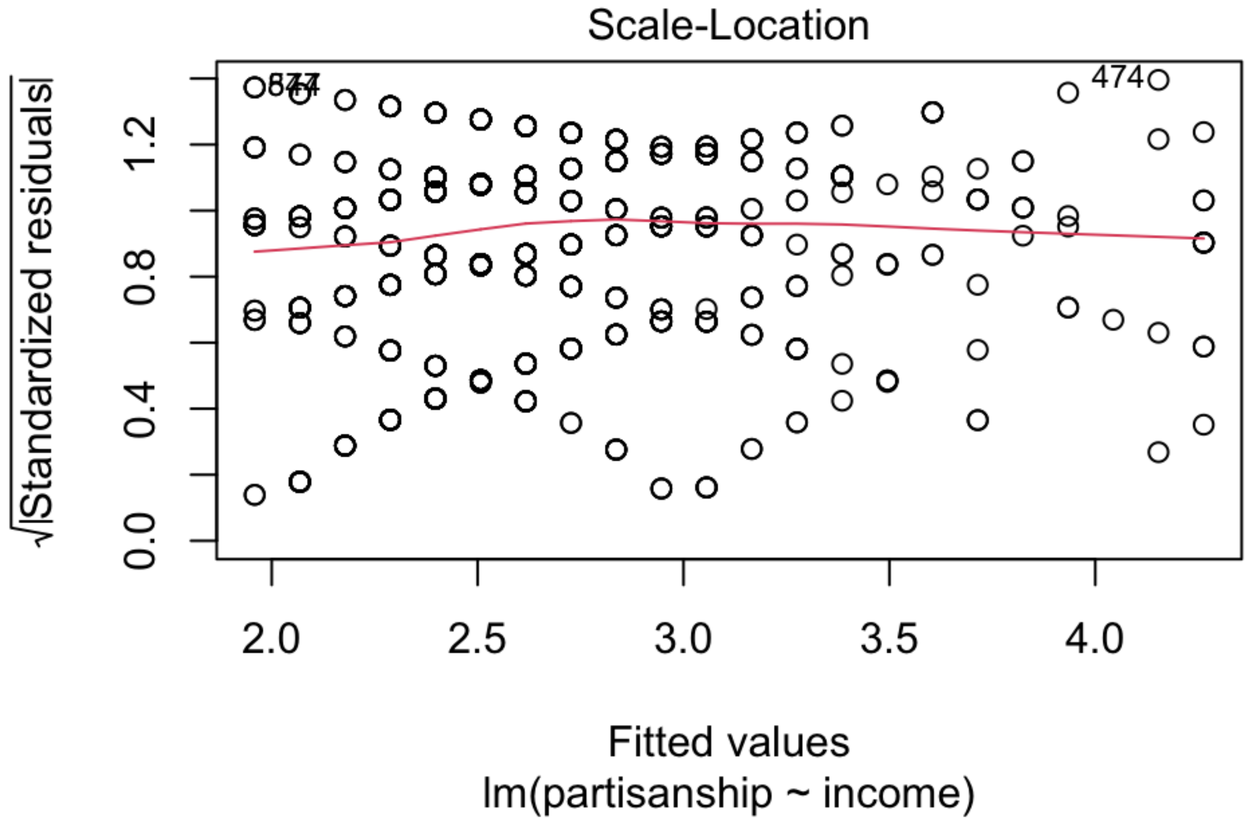
Diagnostics

- Basic diagnostics are very easy!
  - `plot(model)`



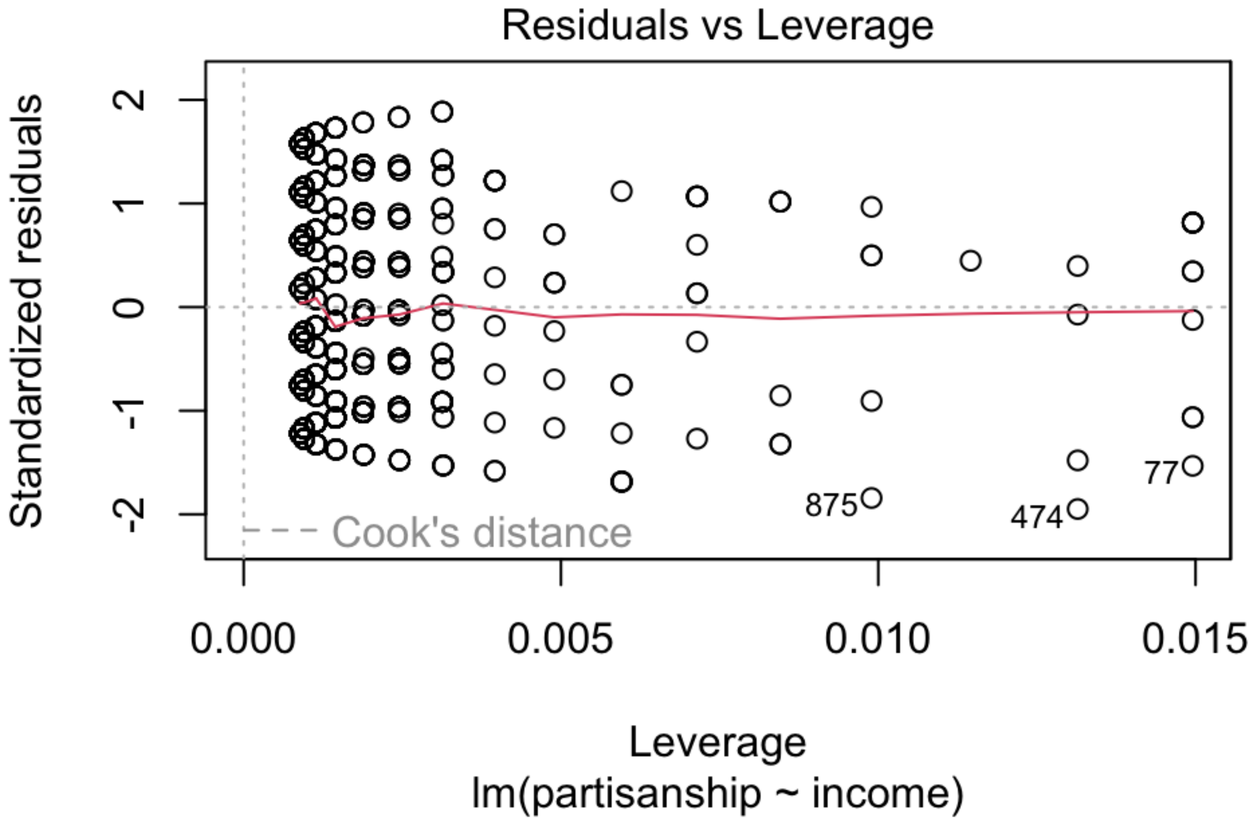
Diagnostics

- Basic diagnostics are very easy!
  - `plot(model)`



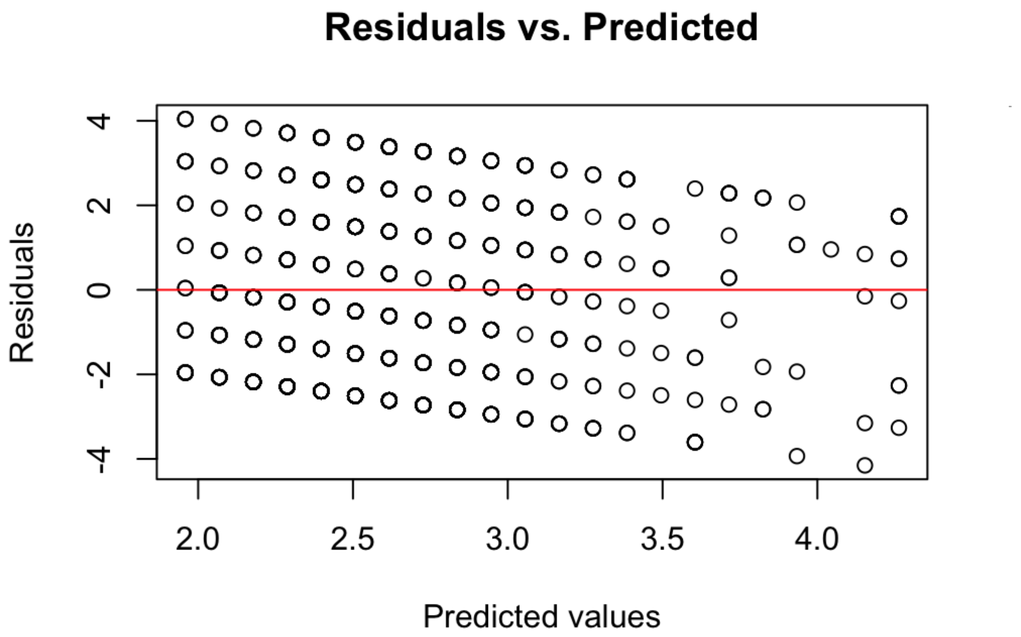
Diagnostics

- Basic diagnostics are very easy!
  - `plot(model)`



Predicted values

- To generate predicted values and residuals manually:
  - `nes2000$pred <- predict(model)`
  - `nes2000$resid <- resid(model)`
- You can then create your own residual plot:
  - `plot(nes2000$pred, nes2000$resid)`
- Or, in slightly cleaner fashion, without using the intermediate variables and labeling:
  - `plot(predict(model), resid(model), xlab = "Predicted values", ylab = "Residuals",main = "Residuals vs. Predicted”)`
  - `abline(h = 0, col = "red")`



Predicted values

- You can also use the predict command for particular values:

```
> predict(model, newdata = data.frame(income = 11))
       1
3.056054
```

- Or multiple values:

```
> range(nes2000$income)
[1]  1 22
> predict(model, newdata = data.frame(income = c(1, 11, 22)))
       1        2        3
1.958602 3.056054 4.263250
```

Multiple Regression is straightforward as well

- Let's say we want to regression partisanship on income and race
  - Defining race as white and nonwhite
  - First, we create the variable

```
table(nes2000$race)
library(tidyverse)
nes2000 <- nes2000 %>%
  mutate(white = if_else(race == 5, 1, 0))
```

- Then, create a model with two right hand side variables

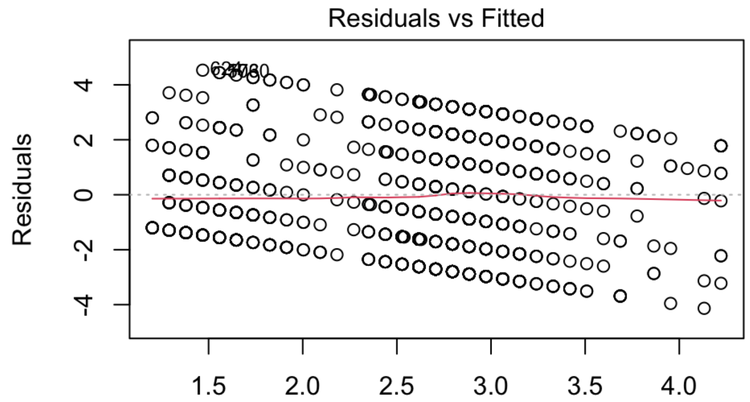
```
model2 <- lm(partisanship ~ income + white, data = nes2000)
```

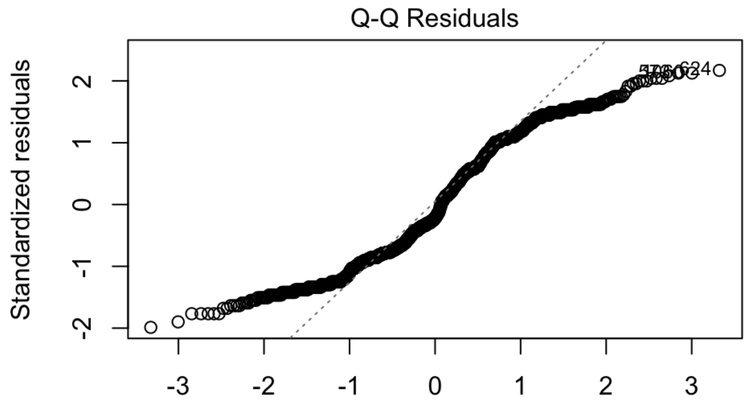
Multiple Regression is straightforward as well

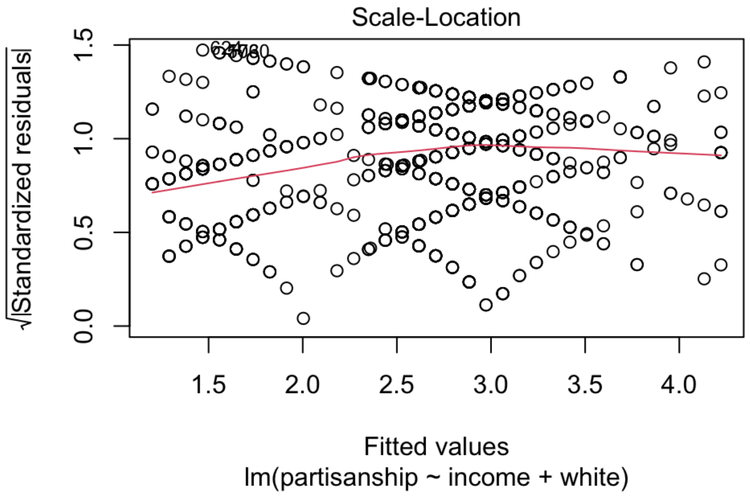
```
> summary(model2)

Call:
lm(formula = partisanship ~ income + white, data = nes2000)

Residuals:
    Min      1Q  Median      3Q     Max
-4.1322 -1.7361 -0.4386  1.9374  4.5313

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.11209    0.16266   6.837 1.33e-11 ***
income       0.08914    0.01675   5.322 1.24e-07 ***
white        1.14821    0.14954   7.678 3.49e-14 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.093 on 1122 degrees of freedom
Multiple R-squared:  0.08409, Adjusted R-squared:  0.08246
F-statistic: 51.51 on 2 and 1122 DF,  p-value: < 2.2e-16
```

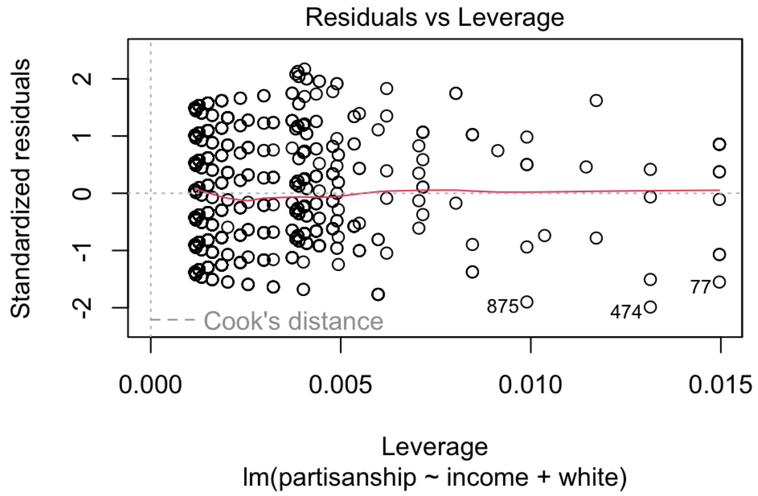
- Everything is still here!
  - Intercept term
  - Slope terms
  - Standard errors
  - T-values
  - P-values
  - “Stars”
  - R^2 (of various kinds)
  - F-statistic
  - Residuals

`plot(model2)` works as well



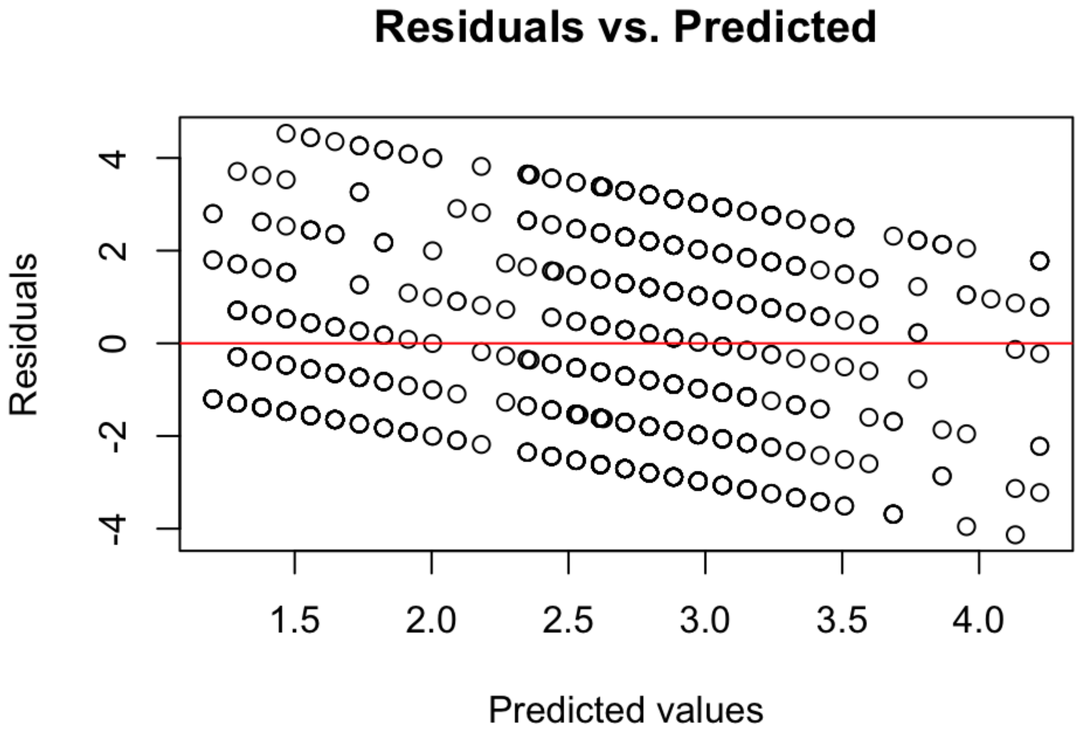






And you can use predict commands in just the same way

- To generate predicted values and residuals manually:
  - `nes2000$pred2 <- predict(model2)`
  - `nes2000$resid2 <- resid(model2)`
- You can then create your own residual plot:
  - `plot(nes2000$pred2, nes2000$resid2)`



- Or, in slightly cleaner fashion, without using the intermediate variables and labeling:
  - `plot(predict(model2), resid(model2), xlab = "Predicted values", ylab = "Residuals",main = "Residuals vs. Predicted”)`
  - `abline(h = 0, col = "red")`

And you can use predict commands in just the same way

- You can also use the predict command for particular values:

```
> predict(model2, newdata = data.frame(income = 11, white = 1))
       1
3.240834
```

- Or multiple values (more usefully this time):

```
> predict(model2, newdata = data.frame(income = c(1, 11, 22, 1,
11, 22), white = c(0, 0, 0, 1,1,1)))
       1        2        3        4        5        6
1.201233 2.092628 3.073163 2.349439 3.240834 4.221368
```

Further extensions: interactions and the `margins()` package

- You can just add multiplicative terms:

```
> model3 <- lm(partisanship ~ income*white, data = nes2000)
> summary(model3)

Call:
lm(formula = partisanship ~ income * white, data = nes2000)

Residuals:
    Min      1Q  Median      3Q     Max
-4.0121 -1.7844 -0.4853  2.0126  4.6141

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)   0.85456    0.27374   3.122  0.00184 **
income        0.13284    0.04095   3.244  0.00121 **
white         1.47004    0.31316   4.694 3.01e-06 ***
income:white -0.05248    0.04487  -1.170  0.24240
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.092 on 1121 degrees of freedom
Multiple R-squared:  0.08521,Adjusted R-squared:  0.08276
F-statistic: 34.81 on 3 and 1121 DF,  p-value: < 2.2e-16
```

Interactive terms are complicated to interpret. margins() is here to help

```
> library(margins)

> marg <- margins(model3)

> summary(marg)

 factor    AME      SE       z       p   lower   upper
 income 0.0926 0.0170 5.4445 0.0000 0.0593 0.1259
  white 1.1032 0.1544 7.1456 0.0000 0.8006 1.4058
```

Interactive terms are complicated to interpret

- **`margins()`** is here to help

```
> library(margins)

> marg <- margins(model3)

> summary(marg)
```

```
 factor    AME      SE       z       p   lower   upper
 income 0.0926 0.0170 5.4445 0.0000 0.0593 0.1259
  white 1.1032 0.1544 7.1456 0.0000 0.8006 1.4058
```

Interactive terms are complicated to interpret

- Alternately, specify the marginal effect of **`income`** at different levels of **`white`**

```
> marg_white <- margins(model3,
    variables = "income",
    at = list(white = c(0,1)))

> summary(marg_white)
 factor  white    AME     SE       z      p  lower  upper
 income 0.0000 0.1328 0.0409 3.2443 0.0012 0.0526 0.2131
 income 1.0000 0.0804 0.0184 4.3784 0.0000 0.0444 0.1163
```

Interactive terms are complicated to interpret

- Interactive terms are complicated!
  - Refer to statistics classes for more theory
  - And the margins() command documentation and vignettes for more programmatic detail
    - https://cran.r-project.org/web/packages/margins/vignettes/Introduction.html

Finally, full dummy variable regression

- Treat categorical variables as factors
  - R automatically drops a baseline category

```
> model4<-lm(partisanship ~ income + as.factor(race), data = nes2000)
> summary(model4)

Call:
lm(formula = partisanship ~ income + as.factor(race), data = nes2000)

Residuals:
    Min      1Q  Median      3Q     Max
-4.0492 -1.8032 -0.2584  1.9476  4.8981

Coefficients:
                  Estimate Std. Error t value Pr(>|t|)
(Intercept)        0.52042    0.19808   2.627 0.008724 **
income             0.08307    0.01663   4.996 6.77e-07 ***
as.factor(race)2   1.52871    0.54843   2.787 0.005403 **
as.factor(race)3   2.76579    0.62212   4.446 9.63e-06 ***
as.factor(race)4   1.09929    0.32048   3.430 0.000625 ***
as.factor(race)5   1.78436    0.19345   9.224  < 2e-16 ***
as.factor(race)6   1.07342    0.37954   2.828 0.004764 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.066 on 1118 degrees of freedom
Multiple R-squared:  0.1105,   Adjusted R-squared:  0.1057
F-statistic: 23.15 on 6 and 1118 DF,  p-value: < 2.2e-16
```

Finally, full dummy variable regression

- Treat categorical variables as factors
  - R automatically drops a baseline category
  - You can choose the baseline manually, too

```
> model4 <- lm(partisanship ~ income + relevel(as.factor(race), ref = "5"),
               data = nes2000)
> summary(model4)

Call:
lm(formula = partisanship ~ income + relevel(as.factor(race),
    ref = "5"), data = nes2000)

Residuals:
    Min      1Q  Median      3Q     Max
-4.0492 -1.8032 -0.2584  1.9476  4.8981

Coefficients:
                                        Estimate Std. Error t value Pr(>|t|)
(Intercept)                              2.30477    0.14060  16.392  < 2e-16 ***
income                                   0.08307    0.01663   4.996 6.77e-07 ***
relevel(as.factor(race), ref = "5")1    -1.78436    0.19345  -9.224  < 2e-16 ***
relevel(as.factor(race), ref = "5")2    -0.25565    0.52155  -0.490   0.6241
relevel(as.factor(race), ref = "5")3     0.98144    0.60118   1.633   0.1029
relevel(as.factor(race), ref = "5")4    -0.68507    0.27646  -2.478   0.0134 *
relevel(as.factor(race), ref = "5")6    -0.71094    0.34271  -2.074   0.0383 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.066 on 1118 degrees of freedom
Multiple R-squared:  0.1105,   Adjusted R-squared:  0.1057
F-statistic: 23.15 on 6 and 1118 DF,  p-value: < 2.2e-16
```

Finally, full dummy variable regression

- Predictions work the same

```
> nes2000$pred4 <- predict(model4)

> nes2000$resid4 <- resid(model4)

> predict(model4, newdata = data.frame(
            income = c(1, 11, 22),
            race = c(1, 2, 3, 4,5,6)))

       1        2        3        4        5        6
0.6034838 2.9628580 5.1136759 1.7027717 3.2185054 3.4212965
```

Logistic Regression is similar: just use the **`glm()`** command

- glm() asks for the regression formula, followed by data, followed by a definition of the model link function - binomial (logit, probit) gaussian, etc

```
> logit_model <- glm(vote ~ income,
                     data   = nes2000,
                     family = binomial(link = "logit"))
> summary(logit_model)

Call:
glm(formula = vote ~ income, family = binomial(link = "logit"),
    data = nes2000)

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) -0.70375    0.12934  -5.441  5.3e-08 ***
income       0.06433    0.01626   3.957  7.6e-05 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 1541.9  on 1124  degrees of freedom
Residual deviance: 1525.8  on 1123  degrees of freedom
AIC: 1529.8

Number of Fisher Scoring iterations: 4
```

```
> probit_model <- glm(vote ~ income,
                      data   = nes2000,
                      family = binomial(link = "probit"))
> summary(probit_model)

Call:
glm(formula = vote ~ income, family = binomial(link = "probit"),
    data = nes2000)

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) -0.43924    0.08003  -5.488 4.06e-08 ***
income       0.04010    0.01007   3.983 6.80e-05 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 1541.9  on 1124  degrees of freedom
Residual deviance: 1525.8  on 1123  degrees of freedom
AIC: 1529.8

Number of Fisher Scoring iterations: 4
```

Generalized linear models

- More specialized models have special libraries
  - `MASS`, `VGAM`, etc
- Similar approach, regardless of model complexity:
  - Run the model, store to object
  - Then use different post-estimation techniques

Models in tidy

- The tidyverse has some particular packages that are useful for tidy model visualization
  - **`broom()`**
    - Cleans somewhat untidy model output (get it, “broom”?) into cleaner objects for tidy visualization and graphing
  - **`margins()`**
    - Invaluable for all kinds of effects plots
- A good reference: Healy, chapter 6, “Work with Models”

Flow Control & Functions

Flow control

- **Flow control**, refers to how you control the order your code executes
  - Or, how you **decide what runs and how often it does**
- Typical flow control components:
  - Conditional statements (if, else) - *run this only if*
    - *You've already used these, like with mutate!*
  - Loops - repeat this thing multiple times
  - Functions - partial off and encapsulate certain elements of code in separate areas
    - *Calling outside scripts is conceptually very similar to this!*

Much of this **should already be familiar**!

- **Flow control**, refers to how you control the order your code executes
  - Or, how you **decide what runs and how often it does**
- Typical flow control components:
  - **Conditional statements (if, else) - run this only if**
    - ***You've already used these, like with mutate!***
  - Loops - repeat this thing multiple times
  - Functions - partial off and encapsulate certain elements of code in separate areas
    - ***Calling outside scripts is conceptually very similar to this!***

Some, however, **is not familiar**

- **Flow control**, refers to how you control the order your code executes
  - Or, how you **decide what runs and how often it does**
- Typical flow control components:
  - **Conditional statements (if, else) - run this only if**
    - ***You've already used these, like with mutate!***
  - **Loops - repeat this thing multiple times**
  - **Functions - partial off and encapsulate certain elements of code in separate areas**
    - **Calling outside scripts is conceptually very similar to this!**

Loops

- Whenever you find yourself doing similar iterated things multiple times in R:
  - Your brain should kick in and say, “there must be an easier way to do this”
- **Loops** are a core programming concept that repeats the same command over and over under certain conditions
  - **for** loop - repeat this thing a number of times
    - **for** loop with **if** commands - conditional decision over series of items
  - **while** loop - repeat this *until* something changes

for loop

- Let's say we want to flag particular values in a dataset

```
for (i in 1:nrow(nes2000)) {
  if (nes2000$income[i] > 20) {
    nes2000$flag[i] <- "High Income"
  } else {
    nes2000$flag[i] <- "Normal"
  }
}
```

- Let's break this apart.

for loop

```
for (i in 1:nrow(nes2000)) {
  if (nes2000$income[i] > 20) {
    nes2000$flag[i] <- "High Income"
  } else {
    nes2000$flag[i] <- "Normal"
  }
}
```

- First, we are identifying ***the kind of loop***

for loop

```
for (i in 1:nrow(nes2000)) {
  if (nes2000$income[i] > 20) {
    nes2000$flag[i] <- "High Income"
  } else {
    nes2000$flag[i] <- "Normal"
  }
}
```

- Second, we identifying ***the range over where this operates***

for loop

```
for (i in 1:nrow(nes2000)) {
  if (nes2000$income[i] > 20) {
    nes2000$flag[i] <- "High Income"
  } else {
    nes2000$flag[i] <- "Normal"
  }
}
```

- Third, we encapsulate ***what happens***

for loop

```
for (i in 1:nrow(nes2000)) {
  if (nes2000$income[i] > 20) {
    nes2000$flag[i] <- "High Income"
  } else {
    nes2000$flag[i] <- "Normal"
  }
}
```

- Fourth, and finally, we identify **what happens**

for loop

```
for (i in 1:nrow(nes2000)) {
  if (nes2000$income[i] > 20) {
    nes2000$flag[i] <- "High Income"
  } else {
    nes2000$flag[i] <- "Normal"
  }
}
```

- Note that what happens has its own nested components
  - *If* a certain value is above 20, *then do* {thing}
  - *else* (when value is not above 20), *do* {other thing}

for loop

```
for (i in 1:nrow(nes2000)) {
  if (nes2000$income[i] > 20) {
    nes2000$flag[i] <- "High Income"
  } else {
    nes2000$flag[i] <- "Normal"
  }
}
```

- Looking at it big picture:
  - Iteratively move with *i* through every row of the dataset
  - Row 1, Row 2, Row 3, etc
  - Do Stuff
  - End when you reach end statement

for loop

```
for (i in 1:nrow(nes2000)) {
  if (nes2000$income[i] > 20) {
    nes2000$flag[i] <- "High Income"
  } else {
    nes2000$flag[i] <- "Normal"
  }
}
```

- The start

id	pid	income
1	4	11
2	4	6
3	1	21
4	3	4

for loop

```
for (i in 1:nrow(nes2000)) {
  if (nes2000$income[i] > 20) {
    nes2000$flag[i] <- "High Income"
  } else {
    nes2000$flag[i] <- "Normal"
  }
}
```

- Things begin with **i=1**
  - search for nes2000$income[i]
    - ie, nes2000$income[1]

id	pid	income
1	4	11
2	4	6
3	1	21
4	3	4

for loop

```
for (i in 1:nrow(nes2000)) {
  if (nes2000$income[i] > 20) {
    nes2000$flag[i] <- "High Income"
  } else {
    nes2000$flag[i] <- "Normal"
  }
}
```

- Now, **if** that number is larger than 20
  - make flag[i]<-“High Income”
- If that number is less than 20
  - make flag[i]<-“Normal"

id	pid	income
1	4	11
2	4	6
3	1	21
4	3	4

for loop

```
for (i in 1:nrow(nes2000)) {
  if (nes2000$income[i] > 20) {
    nes2000$flag[i] <- "High Income"
  } else {
    nes2000$flag[i] <- "Normal"
  }
}
```

- Wait! There is no nes2000$flag

id	pid	income
1	4	11
2	4	6
3	1	21
4	3	4

There is no flag!

But here is where flag[1] would be

for loop

```
for (i in 1:nrow(nes2000)) {
  if (nes2000$income[i] > 20) {
    nes2000$flag[i] <- "High Income"
  } else {
    nes2000$flag[i] <- "Normal"
  }
}
```

- There is no nes2000$flag
  - That's ok!
  - We'll make it
- All done!

id	pid	income	flag
1	4	11	Normal
2	4	6	
3	1	21	
4	3	4	

for loop

```
for (i in 1:nrow(nes2000)) {
  if (nes2000$income[i] > 20) {
    nes2000$flag[i] <- "High Income"
  } else {
    nes2000$flag[i] <- "Normal"
  }
}
```

- Now . . . DO IT AGAIN
  - i iterates to 2
  - search for nes2000$income[i]
    - now, nes2000$income[2]

id	pid	income	flag
1	4	11	Normal
2	4	6	
3	1	21	
4	3	4	

for loop

```
for (i in 1:nrow(nes2000)) {
  if (nes2000$income[i] > 20) {
    nes2000$flag[i] <- "High Income"
  } else {
    nes2000$flag[i] <- "Normal"
  }
}
```

- Rinse and repeat down the column
- **nrow(nes2000)** is 4
  - So we stop when we get to 4

id	pid	income	flag
1	4	11	Normal
2	4	6	Normal
3	1	21	High Income
4	3	4	Normal

i = 1 →

check condition →

run code →

increment i →

repeat until done

for loop

- You could also do this easily with mutate in the tidyverse:

```
nes2000 <- nes2000 |>
  mutate(flag = if_else(income > 20,
"High Income", "Normal"))
```

- And you should usually do that!
  - But loops are much more general controls

id	pid	income	flag
1	4	11	Normal
2	4	6	Normal
3	1	21	High Income
4	3	4	Normal

for loop

- You can, for instance, generate the Fibonnaci sequence for as long as you want:

```
#Let's start by initializing the sequence
fibonacci <- numeric(20) # This makes an empty vector of length 20.
fibonacci[1] <- 1 # The first two numbers are both 1,
fibonacci[2] <- 1 # so we can do those manually.

#Start at 3, go to 20
for(i in 3:20){
# This tells R to make the current number the sum of the last two numbers
  fibonacci[i] <- fibonacci[i - 2] + fibonacci[i - 1]
}
> fibonacci
 [1]    1    1    2    3    5    8   13   21   34   55   89  144  233  377  610
[16]  987 1597 2584 4181 6765
```

while loops

- A while loop continues to do something so long as a certain condition is met.
    - For instance:

```
x <- 1
while (x < 5) {
  print(x)
  x <- x + 1
}
```

    - What would this do?

while loops

- A while loop continues to do something so long as a certain condition is met.
  - For instance:

```
x <- 1
while (x < 5) {
  print(x)
  x <- x + 1
}
```

```
[1] 1
[1] 2
[1] 3
[1] 4
```

  - What would this do?

Or, in Fibonacci land

```
fibonacci <- numeric(20) # This makes an empty vector of length 20. R will expand as necessary.
fibonacci[1] <- 1 # The first two numbers are both 1,
fibonacci[2] <- 1 # so we can do those manually.
i <- 3

while (fibonacci[i - 1] < 100000) {
  fibonacci[i] <- fibonacci[i - 2] + fibonacci[i - 1]
  i <- i + 1
}
> fibonacci
 [1]     1     1     2     3     5     8    13    21    34    55    89
[12]   144   233   377   610   987  1597  2584  4181  6765 10946 17711
[23] 28657 46368 75025 121393
>
```

Functions

Functions and control

- Lots of the time, it's convenient to bracket off certain portions of code to call later
    - We did this initially with the **`source()`** command, which called certain scripts to run
- More formally, we can do similar things by writing our own *functions*
    - An R function is a thing that takes some input and programmatically does math and logic on that input to produce some output

Take, for instance, the imaginary function `bake()`

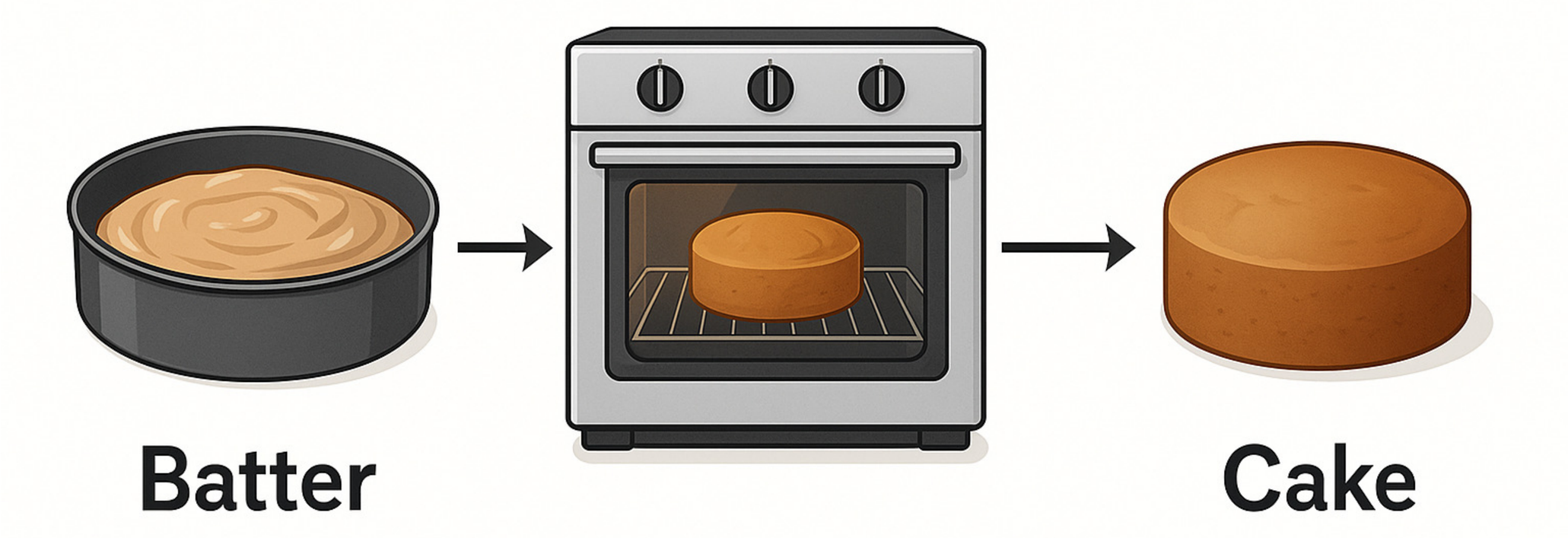
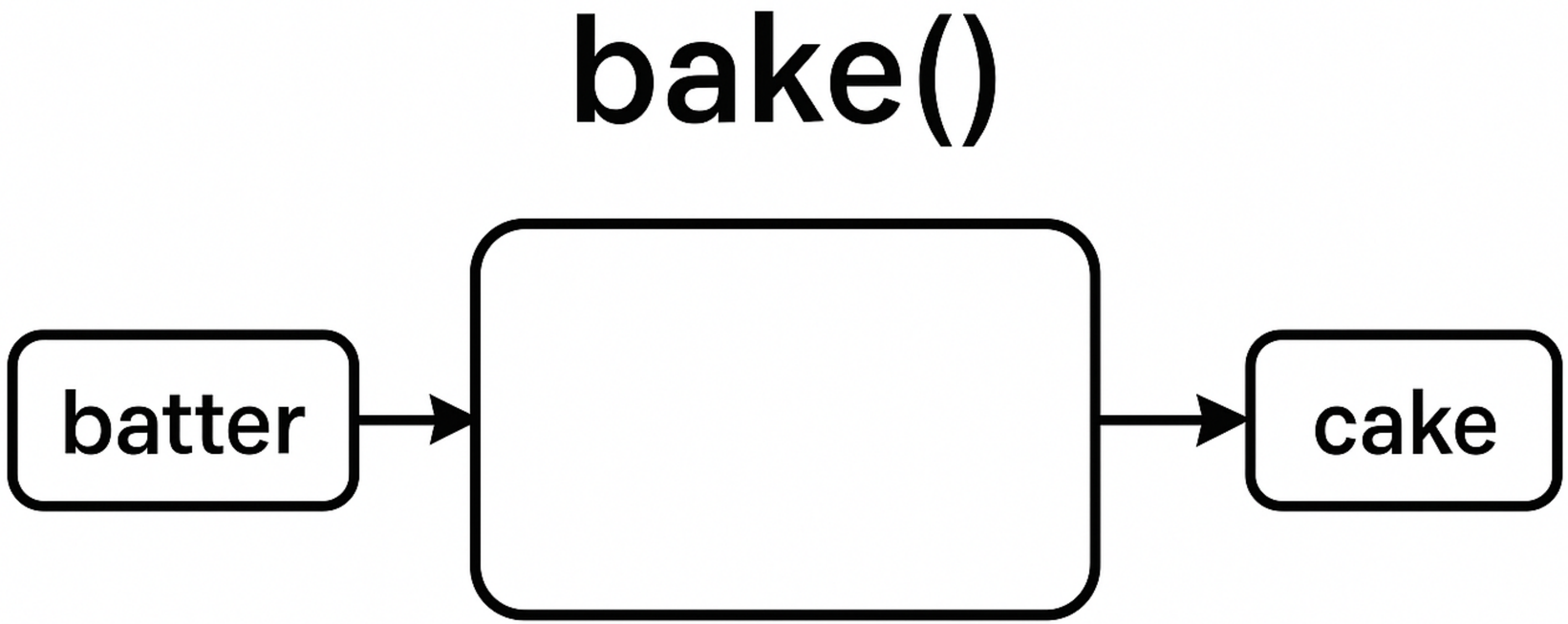


image: ChatGPT

Take, for instance, the imaginary function `bake()`



- the function `bake()` takes the input `batter` and turns it into the output `cake`
  - What happens in the black box: we don't see! we just see the output (delicious delicious cake)

image: ChatGPT

Functions and control

- Everything we do in R is based on functions!
  - **`mutate()`, `vector()`, `lm()`, `glm()`, `t.test()`, `read.csv()`** . . . all functions
- We can also make our own functions!
  - Especially useful for common snippets (or larger than snippets) of code that hold operations we routinely perform

How to make a function

- This is what a function looks like in R!

```
functionname <- function(argument1, argument2, …) {
      math & code stuff
      return(output)
   }
```

How to make a function

- Functions have a few main parts

```
functionname <- function(argument1, argument2, …) {
      math & code stuff
      return(output)
   }
```

- A function is stored as a special object that you name. (It will appear in your environment)

How to make a function

- Functions have a few main parts

```
functionname <- function(argument1, argument2, …) {
      math & code stuff
      return(output)
   }
```

- The object is declared first as a function

How to make a function

- Functions have a few main parts

```
functionname <- function(argument1, argument2, …) {
      math & code stuff
      return(output)
   }
```

- The **inputs** of a function (often called **arguments**) come next.
  - *You can have as many arguments as you need*

How to make a function

- Functions have a few main parts

```
functionname <- function(argument1, argument2, …) {
      math & code stuff
       return(output)
   }
```

- The operations of the function come next, within curly brackets
  - *You can have as many operations as you need. Large functions have a lot of operations!*

How to make a function

- Functions have a few main parts

```
functionname <- function(argument1, argument2, …) {
        math & code stuff
        return(output)
    }
```

- Finally, we define what comes out of the function

How to make a function

- Functions have a few main parts

```
functionname <- function(argument1, argument2, …) {
        math & code stuff
        return(output)
    }
```

- name
- declaration
- arguments/inputs
- code
- output

An example function

```
nummode <- function(x) {
   freq <- table(x)
   return(as.numeric(names(freq)[which.max(freq)]))
}
```

- This function, given a vector, finds and returns the mode of the vector
    - I call it just like any other function (even though I wrote it)

```
> nummode(nes2000$income)
[1] 6
```

Let’s look at what it does

```
nummode <- function(x) {
   freq <- table(x)
   return(as.numeric(names(freq)[which.max(freq)]))
}
```

- Declare its name

Let's look at what it does

```
nummode <- function(x) {
   freq <- table(x)
   return(as.numeric(names(freq)[which.max(freq)]))
}
```

- Declare that it is a function

Let's look at what it does

```
nummode <- function(x) {
   freq <- table(x)
   return(as.numeric(names(freq)[which.max(freq)]))
}
```

- Declare that it takes one argument (named x)

Let's look at what it does

```
nummode <- function(x) {
   freq <- table(x)
   return(as.numeric(names(freq)[which.max(freq)]))
}
```

- Identify the math/code in the black box
  - In this case, what we're doing is making a frequency table in the first line
  - And then, in the second line, finds what the most common occurring value in that frequency table is

Let’s look at what it does

```
nummode <- function(x) {
   freq <- table(x)
   return(as.numeric(names(freq)[which.max(freq)]))
}
```

- name
- declaration
- arguments/inputs
- code
- output

Some additional (optional) tips for functions

- Guardians are checkers to make sure what you're giving the function as an input is what it expects
  - For instance, **`numMode()`** expects a vector. If it doesn't get a vector, it can get cranky in unpredictable ways:

```
> nummode(nes2000)
numeric(0)
> nummode(model2)
Error in table(x) : all arguments must have the same length
```

Show Traceback
Rerun with Debug

```
> nummode("A")
[1] NA
Warning message:
In nummode("A") : NAs introduced by coercion
>
```

So we can add a **guardian** to our function

```
nummodeguard <- function(x) {
  if (!is.vector(x)) {
    stop("Input x must be a vector.")
  }

  freq <- table(x)
  as.numeric(names(freq)[which.max(freq)])
}
```

- Which still produces the same correct output, but gives a nicer warning when needed

```
> nummodeguard(nes2000$income)
[1] 6
> nummodeguard(nes2000)
Error in nummodeguard(nes2000) : Input x must be a vector.
```

Show Traceback
Rerun with Debug

```
> nummodeguard(model2)
Error in nummodeguard(model2) : Input x must be a vector.
```

Show Traceback
Rerun with Debug

```
>
```

First rules for thinking about functions

1. Functions are inputs>>processing>>outputs

First rules for thinking about functions

1. Functions are inputs>>processing>>outputs

2. Function code generally takes three parts

```
myfun <- function(arg1, arg2) {

    # 1. Check / prepare inputs

    # 2. Do the work

    # 3. Return output

}
```

First rules for thinking about functions

1. Functions are inputs>>processing>>outputs
2. Function code generally takes three parts
3. Return values in function can be **implicit** or **explicit**
   - R returns the last evaluated expression automatically

implicit return

```
add <- function(a, b) {
  a + b   # implicitly returned
}
```

explicit return

```
add2 <- function(a, b) {
  return(a + b)   # explicitly returned
}
```

First rules for thinking about functions

1. Functions are inputs>>processing>>outputs
2. Function code generally takes three parts
3. Return values in function can be **implicit** or **explicit**
4. Arguments can be more than just inputs; they can be options. For option arguments, you can argument defaults

Here is a function - **`center()`** - that returns a centered variable value (ie, a value if the variable central tendency was 0). The user can choose median centering or mean centering - default is mean.

```
center <- function(x, type = "mean") {
  if (type == "mean") {
    return(x - mean(x))
  }
  if (type == "median") {
    return(x - median(x))
  }
}
```

First rules for thinking about functions

1. Functions are inputs>>processing>>outputs

2. Function code generally takes three parts

3. Return values in function can be **implicit** or **explicit**

4. Arguments can be more than just inputs; they can be options. For option arguments, you can argument defaults

5. Variables created inside a function belong to the function and disappear afterwards. The only thing that persists is what is returned.

Wrapping Up

- Thursday
  - Practicum 2 coverage
  - Course commentary
  - Work Time
- Next Tuesday
  - Final presentations