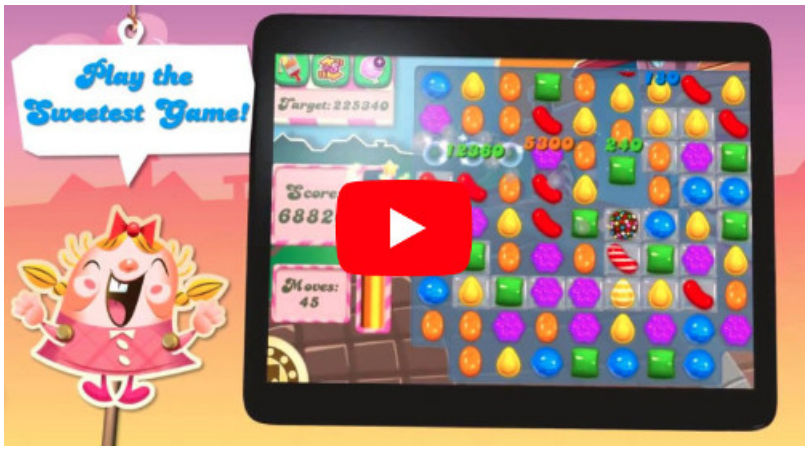
1. Candy Crush Saga

Candy Crush Saga is a hit mobile game developed by King (part of Activision|Blizzard) that is played by millions of people all around the world. The game is structured as a series of levels where players need to match similar candy together to (hopefully) clear the level and keep progressing on the level map. If you are one of the few that haven't played Candy Crush, here's a short demo:



Candy Crush has more than 3000 levels, and new ones are added every week. That is a lot of levels! And with that many levels, it's important to get *level difficulty* just right. Too easy and the game gets boring, too hard and players become frustrated and quit playing.

In this project, we will see how we can use data collected from players to estimate level difficulty. Let's start by loading in the packages we're going to need.

In [82]:

```
# This sets the size of plots to a good default.
options(repr.plot.width = 5, repr.plot.height = 4)

# Loading in packages
library('readr')
library('dplyr')
library('ggplot2')
```

In [83]:

```
library(testthat)
library(IRkernel.testthat)

run_tests({
    test_that("the packages are loaded", {
    expect_true( all(c("package:ggplot2", "package:readr", "package:dplyr") %in% search() ),
        info = "The dplyr, readr and ggplot2 packages should be loaded using import().")
    })
})
```

```
<ProjectReporter>
  Inherits from: <ListReporter>
  Public:
    .context: NULL
    .end_context: function (context)
    .start_context: function (context)
    add_result: function (context, test, result)
    all_tests: environment
    cat_line: function (...)
    cat_tight: function (...)
    clone: function (deep = FALSE)
    current_expectations: environment
    current_file: some name
    current_start_time: 7.746 0.227 5181.007 0.005 0
    dump_test: function (test)
    end_context: function (context)
    end_reporter: function ()
    end_test: function (context, test)
    get_results: function ()
```

```
    initialize: function (...)
    is_full: function ()
    out: 3
    results: environment
    rule: function (...)
    start_context: function (context)
    start_file: function (name)
    start_reporter: function ()
    start_test: function (context, test)
```

2. The data set

The dataset we will use contains one week of data from a sample of players who played Candy Crush back in 2014. The data is also from a single *episode*, that is, a set of 15 levels. It has the following columns:

- **player_id**: a unique player id
- **dt**: the date
- **level**: the level number within the episode, from 1 to 15.
- **num_attempts**: number of level attempts for the player on that level and date.
- **num_success**: number of level attempts that resulted in a success/win for the player on that level and date.

The granularity of the dataset is player, date, and level. That is, there is a row for every player, day, and level recording the total number of attempts and how many of those resulted in a win.

Now, let's load in the dataset and take a look at the first couple of rows.

In [84]:

```
# Reading in the data
data <- read_csv("datasets/candy_crush.csv")

# Printing out the first couple of rows
head(data,2)
```

```
Parsed with column specification:
cols(
  player_id = col_character(),
  dt = col_date(format = ""),
  level = col_double(),
  num_attempts = col_double(),
  num_success = col_double()
)
```

player_id	dt	level	num_attempts	num_success
6dd5af4c7228fa353d505767143f5815	2014-01-04	4	3	1
c7ec97c39349ab7e4d39b4f74062ec13	2014-01-01	8	4	1

In [85]:

```
library(tidyverse)

run_tests({
    test_that("data is read in correctly", {
        correct_data <- read_csv("datasets/candy_crush.csv")
        expect_equal(correct_data, data,
            info = "data should countain datasets/candy_crush.csv read in using read_csv")
        })
})
```

```
<ProjectReporter>
  Inherits from: <ListReporter>
  Public:
    .context: NULL
    .end_context: function (context)
    .start_context: function (context)
    add_result: function (context, test, result)
    all_tests: environment
    cat_line: function (...)
    cat_tight: function (...)
```

```
    clone: function (deep = FALSE)
    current_expectations: environment
    current_file: some name
    current_start_time: 7.793 0.231 5181.058 0.005 0
    dump_test: function (test)
    end_context: function (context)
    end_reporter: function ()
    end_test: function (context, test)
    get_results: function ()
    initialize: function (...)
    is_full: function ()
    out: 3
    results: environment
    rule: function (...)
    start_context: function (context)
    start_file: function (name)
    start_reporter: function ()
    start_test: function (context, test)
```

3. Checking the data set

Now that we have loaded the dataset let's count how many players we have in the sample and how many days worth of data we have.

In [86]:

```
print("Number of players:")
length(unique(data$player_id))

print("Period for which we have data:")
range(data$dt)
```

```
[1] "Number of players:"
```

6814

```
[1] "Period for which we have data:"
```

2014-01-01 2014-01-07

In [87]:

```
run_tests({
    test_that("nothing", {
        expect_true(TRUE, info = "")
    })
})
```

```
<ProjectReporter>
  Inherits from: <ListReporter>
  Public:
    .context: NULL
    .end_context: function (context)
    .start_context: function (context)
    add_result: function (context, test, result)
    all_tests: environment
    cat_line: function (...)
    cat_tight: function (...)
    clone: function (deep = FALSE)
    current_expectations: environment
    current_file: some name
    current_start_time: 7.864 0.231 5181.127 0.005 0
    dump_test: function (test)
    end_context: function (context)
    end_reporter: function ()
    end_test: function (context, test)
    get_results: function ()
    initialize: function (...)
    is_full: function ()
    out: 3
```

```
    results: environment
    rule: function (...)
    start_context: function (context)
    start_file: function (name)
    start_reporter: function ()
    start_test: function (context, test)
```

4. Computing level difficulty

Within each Candy Crush episode, there is a mix of easier and tougher levels. Luck and individual skill make the number of attempts required to pass a level different from player to player. The assumption is that difficult levels require more attempts on average than easier ones. That is, *the harder* a level is, *the lower* the probability to pass that level in a single attempt is.

A simple approach to model this probability is as a Bernoulli process; as a binary outcome (you either win or lose) characterized by a single parameter p_{win}: the probability of winning the level in a single attempt. This probability can be estimated for each level as:

$$p_{win} = \frac{\sum wins}{\sum attempts}$$

For example, let's say a level has been played 10 times and 2 of those attempts ended up in a victory. Then the probability of winning in a single attempt would be p_{win} = 2 / 10 = 20%.

Now, let's compute the difficulty p_{win} separately for each of the 15 levels.

In [88]:

```
# Calculating level difficulty
difficulty <- data %>%
 group_by(level) %>%
 summarise(attempts = sum(num_attempts), wins = sum(num_success)) %>%
 mutate(p_win = wins/attempts)


# Printing out the level difficulty
difficulty
```

level	attempts	wins	p_win
1	1322	818	0.61875946
2	1285	666	0.51828794
3	1546	662	0.42820181
4	1893	705	0.37242472
5	6937	634	0.09139397
6	1591	668	0.41986172
7	4526	614	0.13566063
8	15816	641	0.04052858
9	8241	670	0.08130081
10	3282	617	0.18799512
11	5575	603	0.10816143
12	6868	659	0.09595224
13	1327	686	0.51695554
14	2772	777	0.28030303
15	30374	1157	0.03809179

In [89]:

```
run_tests({
    test_that("p_win is calculated correctly", {
        correct_difficulty <- data %>%
            group_by(level) %>%
            summarise(attempts = sum(num_attempts), wins = sum(num_success)) %>%
```

```
            mutate(p_win = wins / attempts)
        expect_equal(correct_difficulty$p_win, difficulty$p_win,
            info = "difficulty$p_win should be estimated probability to pass each level in a single
attempt")
        })
})
```

```
<ProjectReporter>
  Inherits from: <ListReporter>
  Public:
    .context: NULL
    .end_context: function (context)
    .start_context: function (context)
    add_result: function (context, test, result)
    all_tests: environment
    cat_line: function (...)
    cat_tight: function (...)
    clone: function (deep = FALSE)
    current_expectations: environment
    current_file: some name
    current_start_time: 7.905 0.234 5181.171 0.005 0
    dump_test: function (test)
    end_context: function (context)
    end_reporter: function ()
    end_test: function (context, test)
    get_results: function ()
    initialize: function (...)
    is_full: function ()
    out: 3
    results: environment
    rule: function (...)
    start_context: function (context)
    start_file: function (name)
    start_reporter: function ()
    start_test: function (context, test)
```

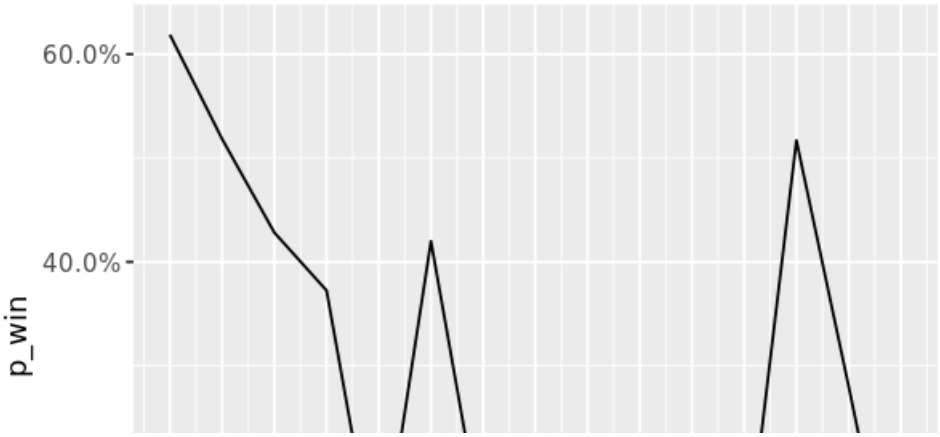
5. Plotting difficulty profile

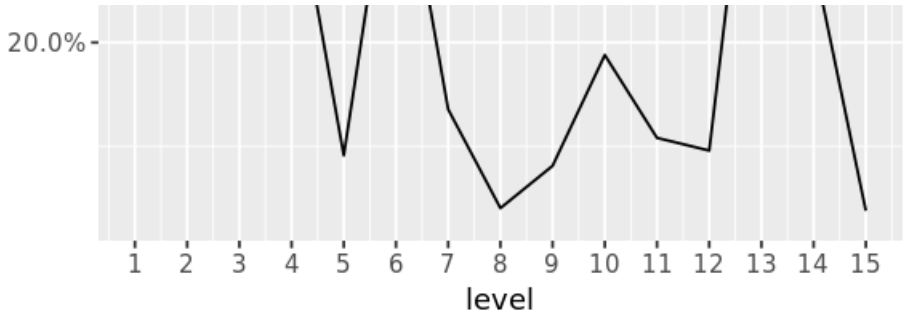
Great! We now have the difficulty for all the 15 levels in the episode. Keep in mind that, as we measure difficulty as the probability to pass a level in a single attempt, a *lower* value (a smaller probability of winning the level) implies a *higher* level difficulty.

Now that we have the difficulty of the episode we should plot it. Let's plot a line graph with the levels on the X-axis and the difficulty (p_{win}) on the Y-axis. We call this plot the *difficulty profile* of the episode.

In [90]:

```
# Plotting the level difficulty profile
difficulty %>%
  ggplot(aes(x = level, y = p_win)) +
  geom_line() +
  scale_x_continuous(breaks = 1:15) +
  scale_y_continuous(label = scales::percent)
```





In [91]:

```
run_tests({
    test_that("the student plotted a ggplot", {
    expect_true('ggplot' %in% class(last_plot()),
        info = "You should plot difficulty using ggplot.")
    })
})
```

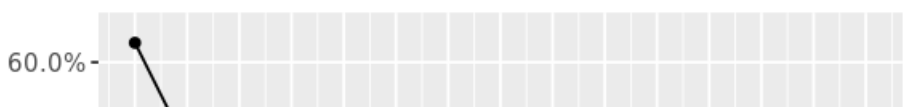
```
<ProjectReporter>
  Inherits from: <ListReporter>
  Public:
    .context: NULL
    .end_context: function (context)
    .start_context: function (context)
    add_result: function (context, test, result)
    all_tests: environment
    cat_line: function (...)
    cat_tight: function (...)
    clone: function (deep = FALSE)
    current_expectations: environment
    current_file: some name
    current_start_time: 8.116 0.235 5181.381 0.005 0
    dump_test: function (test)
    end_context: function (context)
    end_reporter: function ()
    end_test: function (context, test)
    get_results: function ()
    initialize: function (...)
    is_full: function ()
    out: 3
    results: environment
    rule: function (...)
    start_context: function (context)
    start_file: function (name)
    start_reporter: function ()
    start_test: function (context, test)
```

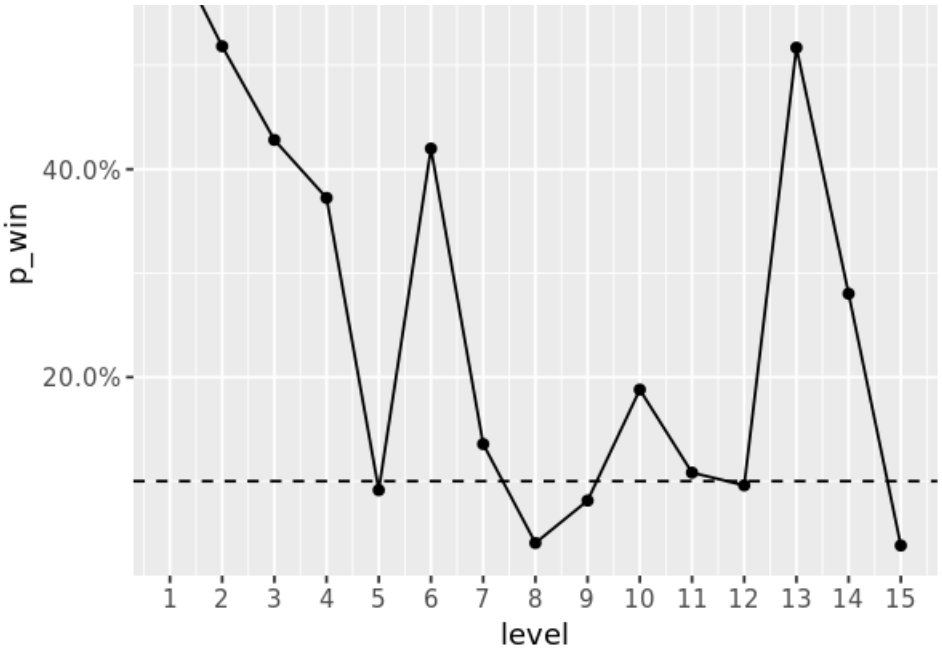
6. Spotting hard levels

What constitutes a *hard* level is subjective. However, to keep things simple, we could define a threshold of difficulty, say 10%, and label levels with p_{win} < 10% as *hard*. It's relatively easy to spot these hard levels on the plot, but we can make the plot more friendly by explicitly highlighting the hard levels.

In [92]:

```
# Adding points and a dashed line
difficulty %>%
ggplot(aes(x = level, y = p_win)) +
  geom_line() +
  geom_point() +
  scale_x_continuous(breaks = 1:15) +
  scale_y_continuous(label = scales::percent) +
  geom_hline(yintercept = 0.1, linetype = 'dashed')
```





In [93]:

```
run_tests({
    plot_layers <- sapply(last_plot()$layers, function(layer)  class(layer$geom)[1])
    test_that("the student has plotted lines, points and a hline", {
    expect_true(all(c('GeomLine', 'GeomPoint', 'GeomHline') %in%  plot_layers),
        info = "The plot should include lines between the datapoints, points at the datapoints and
a horisontal line.")
    })
})
```

```
<ProjectReporter>
  Inherits from: <ListReporter>
  Public:
    .context: NULL
    .end_context: function (context)
    .start_context: function (context)
    add_result: function (context, test, result)
    all_tests: environment
    cat_line: function (...)
    cat_tight: function (...)
    clone: function (deep = FALSE)
    current_expectations: environment
    current_file: some name
    current_start_time: 8.349 0.239 5181.618 0.005 0
    dump_test: function (test)
    end_context: function (context)
    end_reporter: function ()
    end_test: function (context, test)
    get_results: function ()
    initialize: function (...)
    is_full: function ()
    out: 3
    results: environment
    rule: function (...)
    start_context: function (context)
    start_file: function (name)
    start_reporter: function ()
    start_test: function (context, test)
```

7. Computing uncertainty

As Data Scientists we should always report some measure of the uncertainty of any provided numbers. Maybe tomorrow, another sample will give us slightly different values for the difficulties? Here we will simply use the *Standard error* as a measure of uncertainty:

$$\sigma_{error} \approx \frac{\sigma_{sample}}{\sqrt{n}}$$

$$\sqrt{n}$$

Here n is the number of datapoints and σ_{sample} is the sample standard deviation. For a Bernoulli process, the sample standard deviation is:

$$\sigma_{sample} = \sqrt{p_{win}(1 - p_{win})}$$

Therefore, we can calculate the standard error like this:

$$\sigma_{error} \approx \sqrt{\frac{p_{win}(1 - p_{win})}{n}}$$

We already have all we need in the `difficulty` data frame! Every level has been played n number of times and we have their difficulty p_{win}. Now, let's calculate the standard error for each level.

In [94]:

```
# Computing the standard error of p_win for each level
difficulty <- difficulty %>%
   mutate( error = sqrt(p_win * (1 - p_win) / attempts) )
difficulty
```

level	attempts	wins	p_win	error
1	1322	818	0.61875946	0.013358101
2	1285	666	0.51828794	0.013938876
3	1546	662	0.42820181	0.012584643
4	1893	705	0.37242472	0.011111607
5	6937	634	0.09139397	0.003459878
6	1591	668	0.41986172	0.012373251
7	4526	614	0.13566063	0.005089930
8	15816	641	0.04052858	0.001568008
9	8241	670	0.08130081	0.003010538
10	3282	617	0.18799512	0.006819983
11	5575	603	0.10816143	0.004159651
12	6868	659	0.09595224	0.003553924
13	1327	686	0.51695554	0.013717807
14	2772	777	0.28030303	0.008530846
15	30374	1157	0.03809179	0.001098327

In [95]:

```
run_tests({
    test_that("error is correct", {
        correct_difficulty <- difficulty %>%
            mutate(error = sqrt(p_win * (1 - p_win) / attempts))
        expect_equal(correct_difficulty$error, difficulty$error,
            info = "difficulty$error should be calculated as sqrt(p_win * (1 - p_win) / attempts)")
    })
})
```

```
<ProjectReporter>
  Inherits from: <ListReporter>
  Public:
    .context: NULL
    .end_context: function (context)
    .start_context: function (context)
    add_result: function (context, test, result)
    all_tests: environment
    cat_line: function (...)
    cat_tight: function (...)
```

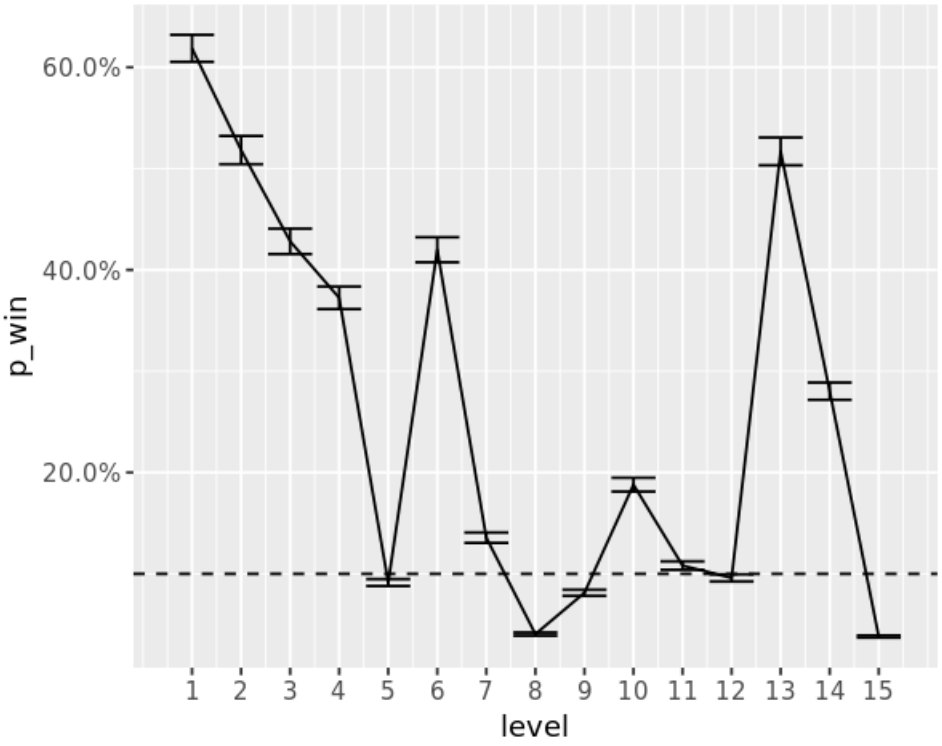
```
clone: function (deep = FALSE)
current_expectations: environment
current_file: some name
current_start_time: 8.406 0.239 5181.674 0.005 0
dump_test: function (test)
end_context: function (context)
end_reporter: function ()
end_test: function (context, test)
get_results: function ()
initialize: function (...)
is_full: function ()
out: 3
results: environment
rule: function (...)
start_context: function (context)
start_file: function (name)
start_reporter: function ()
start_test: function (context, test)
```

8. Showing uncertainty

Now that we have a measure of uncertainty for each levels' difficulty estimate let's use *error bars* to show this uncertainty in the plot. We will set the length of the error bars to one standard error. The upper limit and the lower limit of each error bar should then be p_{win} + σ_{error} and p_{win} - σ_{error}, respectively.

In [96]:

```
# Adding standard error bars
difficulty %>%
ggplot(aes(x = level, y = p_win)) +
  geom_line() +
  scale_x_continuous(breaks = 1:15) +
  scale_y_continuous(label = scales::percent) +
  geom_hline(yintercept = 0.1, linetype = 'dashed') +
  geom_errorbar( aes(ymin =p_win - error , ymax = p_win + error) )
```



In [97]:

```
run_tests({
    plot_layers <- sapply(last_plot()$layers, function(layer)  class(layer$geom)[1])
    test_that("the student has plotted lines, points and a hline", {
    expect_true("GeomErrorbar" %in%  plot_layers,
        info = "The plot should include error bats using geom_errorbar.")
```

```
    })
})
```

```
<ProjectReporter>
  Inherits from: <ListReporter>
  Public:
    .context: NULL
    .end_context: function (context)
    .start_context: function (context)
    add_result: function (context, test, result)
    all_tests: environment
    cat_line: function (...)
    cat_tight: function (...)
    clone: function (deep = FALSE)
    current_expectations: environment
    current_file: some name
    current_start_time: 8.689 0.239 5181.958 0.005 0
    dump_test: function (test)
    end_context: function (context)
    end_reporter: function ()
    end_test: function (context, test)
    get_results: function ()
    initialize: function (...)
    is_full: function ()
    out: 3
    results: environment
    rule: function (...)
    start_context: function (context)
    start_file: function (name)
    start_reporter: function ()
    start_test: function (context, test)
```

9. A final metric

It looks like our difficulty estimates are pretty precise! Using this plot, a level designer can quickly spot where the hard levels are and also see if there seems to be too many hard levels in the episode.

One question a level designer might ask is: "How likely is it that a player will complete the episode without losing a single time?" Let's calculate this using the estimated level difficulties!

In [98]:

```
# The probability of completing the episode without losing a single time
p <- prod(difficulty$p_win)

# Printing it out
p
```

9.44714093448606e-12

In [99]:

```
run_tests({
    test_that("p is correct", {
        correct_p <- prod(difficulty$p_win)
        expect_equal(correct_p, p,
            info = "p should be calculated as the product of difficulty$p_win .")
    })
})
```

```
<ProjectReporter>
  Inherits from: <ListReporter>
  Public:
    .context: NULL
    .end_context: function (context)
    .start_context: function (context)
    add_result: function (context, test, result)
    all_tests: environment
    cat_line: function (...)
    cat_tight: function (...)
    clone: function (deep = FALSE)
```

```
    current_expectations: environment
    current_file: some name
    current_start_time: 8.726 0.242 5181.998 0.005 0
    dump_test: function (test)
    end_context: function (context)
    end_reporter: function ()
    end_test: function (context, test)
    get_results: function ()
    initialize: function (...)
    is_full: function ()
    out: 3
    results: environment
    rule: function (...)
    start_context: function (context)
    start_file: function (name)
    start_reporter: function ()
    start_test: function (context, test)
```

10. Should our level designer worry?

Given the probability we just calculated, should our level designer worry about that a lot of players might complete the episode in one attempt?

In [100]:

```
# Should our level designer worry about that a lot of
# players will complete the episode in one attempt?
should_the_designer_worry = FALSE # TRUE / FALSE
```

In [101]:

```
run_tests({
    test_that("should_the_designer_worry is FALSE", {
    expect_false(should_the_designer_worry,
        info = "The probability is really small, so I don't think the designer should worry that m
uch...")
    })
})
```

```
<ProjectReporter>
  Inherits from: <ListReporter>
  Public:
    .context: NULL
    .end_context: function (context)
    .start_context: function (context)
    add_result: function (context, test, result)
    all_tests: environment
    cat_line: function (...)
    cat_tight: function (...)
    clone: function (deep = FALSE)
    current_expectations: environment
    current_file: some name
    current_start_time: 8.764 0.243 5182.035 0.005 0
    dump_test: function (test)
    end_context: function (context)
    end_reporter: function ()
    end_test: function (context, test)
    get_results: function ()
    initialize: function (...)
    is_full: function ()
    out: 3
    results: environment
    rule: function (...)
    start_context: function (context)
    start_file: function (name)
    start_reporter: function ()
    start_test: function (context, test)
```