Stat 151 Exam 1 Key

Instructions

- The exam is designed to assess both your fluency and your competency with the skills in Stat 151.
  - You may not finish within 2 hours – that is ok.
  - There is redundancy built in
- There are two sections on this exam.
  - Each section is broken down into a number of skills that should be attempted in order.
  - Attempt Section 1 before you start Section 2
  - Complete a section in one language before attempting it in the other.
  - Both sections are designed to be equally easy in R or Python.
- If you get stuck:
  - Two pre-written hints for each section/skill combination can be obtained for 30% of the value of that question each.
  - If the hints do not help, you can get a solution that will allow you to attempt the next skill, but you will not receive credit for the skill.

Allowed resources:

- notes
- homework
- Statistical Computing in R and Python (the course textbook)

Use of any other resources will result in a 0 on the exam and an academic misconduct report.

Your code is expected to run on my machine when I grade this exam. You will receive reduced credit for code which does not work correctly, whether or not it produces explicit errors.

1 Prime Factorization

In this problem, you will work towards building code that will decompose a number into its prime factors.

We'll start by installing a package that helps work with prime numbers… `primes` in R, `SymPy` in Python.

1.1 Skill: Install Packages

R

Write code to install the `primes` package in R. Run the code.

```
install.packages("primes")
```

Python

Write code to install the `SymPy` package in Python. If you choose to install the package via the terminal, place your code in the bash chunk; otherwise, place your code in the python chunk.

```
pip install sympy
```

```
%pip install sympy
```

Thinking Criticially

You must complete this question regardless of the language used above.

1. Add `, eval = F` to the code chunk header to stop that code from running every time the document is executed.
2. List at least one additional way to keep the code in the quarto document but prevent it from running.

Comment the line of code out using `#` at the front of the line.

1.2 Skill: Loading packages

Load the packages you just installed in R and Python using the chunks below.

R

```r
library(primes)
```

Python

Hint: SymPy should be typed as `sympy` when loading the package. You will only need the `primerange` function in SymPy.

```python
from sympy import primerange
# import sympy as sp
# import sympy
```

1.3 Skill: Using prewritten functions

R

Use the `generate_primes(min, max)` function to generate all the primes between 1 and 47, inclusive. Store these numbers in a variable called `myprimes`.

```
myprimes <- generate_primes(1, 47)
```

Python

Use the `primerange(a, b)` function in `SymPy` to get all primes between 1 and 47, inclusive. Store these numbers in a variable called `myprimes`.

You may have to convert the output of `primerange` to a `list` by wrapping the function call in `list()`. It may be convenient to then convert your list into a numpy array using `np.array(list(...))`. If you choose to do this, you will also need to import the numpy library.

```
# import numpy as np
# myprimes = np.array(list(primerange(1, 48)))
myprimes = list(primerange(1, 48)) # without numpy
```

Comparison

Do the function parameters you used in R and python differ? Why?

In Python, intervals are half-open, that is, `primerange(1, 47)` would not include 47, so it is necessary to use `primerange(1, 48)` to get 47, which is prime. In R, intervals are closed, so `generate_primes(1, 47)` includes 47 within the acceptable interval.

1.4 Skill: Indexing and Subsets

Using your `primes` variables (in both R and python), determine which primes are a factor of a second variable, `x`. You may want to test your code on values like `x=18`, `x=25`, `x = 34`, `x=43`, Your code should output only the prime factors of `x`.

> **Reminder: Modular Division**
>
> Remember that modular division gives you the remainder when dividing a number by another number. `x %% 3` (R) or `x % 3` (Python) will give you the remainder when `x` is divided by 3.
> If `x` is an integer and the divisor is an integer, the result of modular division will also be an integer.

R

```
x <- 43
myprimes[x%%myprimes == 0]
```

```
[1] 43
```

Python

```
x = 18
[i for i in myprimes if x%i==0] # without numpy
# myprimes[x%myprimes == 0] # with numpy
```

```
[2, 3]
```

1.5 Skill: Writing Functions

In the previous section, you determined which primes were factors of a given number. Of course, it is possible to have a number which has multiple occurrences of the same prime factor.

In R and python, write a function, `prime_n(x, prime)`, which will determine how many times a single, pre-specified prime number `prime` can be evenly divided into a number `x`. You may want to calculate an upper bound for this possibility to help you in your search. Your function should take parameters `x` and `prime` and return the integer number of times `prime` occurs in the prime factorization of `x`.

> **i** Reminder: Logs with different bases
>
> The operation $\log_a(x) = b$ is defined the number b such that $a^b = x$ for fixed a and x. You can use this information to determine the maximum number of times a factor could be repeated when calculating the prime factorization of a number.
> You will need to load the `math` library in Python to access the `math.log(a, Base)` function. In R, the `log(x, base)` function is built in.
> It may also be useful to know that the `floor()` function (R) and `math.floor()` function (Python) will round down to the integer below a value.

R

```
prime_n <- function(x, prime) {
  max_n <- floor(log(x, base =  prime))
  prime_seq <- prime^(1:max_n)
  which.max(prime_seq[x%%prime_seq == 0])
}

prime_n(18, 3)
```

```
[1] 2
```

```
prime_n(32, 2)
```

```
[1] 5
```

A number of you handled this using a while loop – which is a great way to do it, and doesn't require the `log` hint I gave you.

```
prime_n <- function(x, prime) {
  times <- 0
  while(x%%prime == 0) {
    x <- x/prime # Divide x by prime
    times <- times + 1 # Increase times to account for that
  }

  if(times == 0) {
    return(NULL)
  } else {
    return(times)
  }
}

prime_n(18, 3)
```

```
[1] 2
```

```
prime_n(32, 2)
```

```
[1] 5
```

Python

```
import math
import numpy as np
def prime_n(x, prime):
  max_n = math.floor(math.log(x, prime))
  exp_opts = np.array(range(1, max_n+1))
  prime_seq = prime**(exp_opts)
  ans = max(exp_opts[x%prime_seq == 0])
  return ans


prime_n(18, 3)
```

```
2
```

```
prime_n(32, 2)
```

5

```
import math
import numpy as np
def prime_n(x, prime):
  times=0
  while(x%prime==0):
    x=x/prime
    times = times + 1

  if(times != 0):
    return times

  return 'Not a factor'


prime_n(18, 3)
```

2

```
prime_n(32, 2)
```

5

1.6 Skill: Data Frames, Loops

Putting the pieces together, use your function and your list of prime factors to determine the prime factorization of a given number. Store your factorization as a data frame with two columns: factor and power, where factor contains the factor and power contains the number of times that factor appears in the prime factorization.

For instance, your result for `x=18` should be a data frame that looks like this:

factor	power
2	1
3	2

Hint:

- R: `rbind(df, row)` will add `row` to the bottom of `df` if `df` is a data frame
- Python: `pandas.concat([df, row])` will add `row` to the bottom of `df` if `df` is a data frame.

Planning

Using the provided scratch paper (please put your name at the top), sketch a basic program flow map that shows how the code you've already written fits together to solve this problem. Identify any bits of logic you need to write to solve the problem.

My solution is sketched out on sheet _____

R

```r
x = 56

factors <- myprimes[x%%myprimes==0]
my_factorization <- data.frame()
for(i in factors) {
  times <- prime_n(x, i)
  my_factorization <- rbind(my_factorization,
                            data.frame(factor = i, power = times))
}

my_factorization
```

```
  factor power
1      2     3
2      7     1
```

Python

```python
import pandas as pd
x = 56

factors = [i for i in myprimes if x%i==0]
my_factorization = pd.DataFrame(columns=['factor', 'power'])
for i in factors:
  times = prime_n(x, i)
  new_row = pd.DataFrame([[i, times]], columns=['factor', 'power'])
  my_factorization = pd.concat([my_factorization, new_row], ignore_index=True)

my_factorization
```

```
  factor power
0      2     3
1      7     1
```

1.7 Skill: String Operations

Take the data frame you created in the previous problem and write a `format_factorization(df)` function that will output the results of that data frame as a string, so that the data frame containing the prime factorization of 18 that is shown above would return "2^1 x 3^2".

Hint:

- Python: in a DataFrame, you can convert the whole column to a string using `df.colname.astype("str")` (replace df, colname with appropriate data frame name and column name)

R

```
format_factorization <- function(df) {
  stopifnot("factor" %in% names(df), "power" %in% names(df))

  paste(paste0(df$factor, "^", df$power), collapse = " x ")
}
```

Python

```
def format_factorization(df):
  mystr = df.factor.astype("str") + "^" + df.power.astype("str")
  return " x ".join(mystr)

format_factorization(my_factorization)
```

```
'2^3 x 7^1'
```

1.8 Skill: Control Statements

Take the code you wrote in the previous part and use it to create a function `prime_factorize` that will return the prime factorization of a number. If the number provided is a prime, your function should return " is prime" instead of returning that the factorization is ^1 (which is not as clear).

Planning

What modifications will you need to make to handle any number? e.g. what if the number is greater than 47?

Need to list out all primes which are strictly less than the number, using the original functions in `primes` and `sympy`.

What modifications will you need to make to handle prime numbers?

Add an if statement detecting whether the number is prime (has no prime factors $<$ itself) and output the prime number statement instead of the factorization.

How can you use previously written code and functions to accomplish this task?

- Obtain primes using `primerange` and `generate_primes`
- Determine which primes are factors using subsetting and modular division
- If a number has no prime factors, return the prime message
- Otherwise, use `prime_n` function to determine how many times each prime is a factor
- Use this information to build a data frame
- Use the `prime_factorization` function to return the prime factorization

What additional code do you need to write?

Code to obtain a generic number of primes

If statement to test for primeness

If you sketched anything for this problem out on scratch paper, please give me a page number to look at.

My solution is sketched out on sheet _____

R

```r
prime_factorize <- function(x) {
  stopifnot(is.numeric(x))
  # Use prewritten function, but don't get x --
  #   get only primes less than x
  myprimes <- generate_primes(2, x-1)
  # Check whether primes are factors
  factors <- myprimes[x%%myprimes==0]

  # Test whether x is prime
  if (length(factors) == 0) {
    return(paste(x, "is a prime number"))
  }

  # This code only runs if x is not prime
  my_factorization <- data.frame()
  for(i in factors) {
    times <- prime_n(x, i)
    my_factorization <- rbind(my_factorization,
                              data.frame(factor = i,
                                         power = times))
  }

  # Return the formatted factorization
  return(format_factorization(my_factorization))
}
```

```r
prime_factorize(34253)
```

```
[1] "34253 is a prime number"
```

```r
prime_factorize(prod(2:10))
```

```
[1] "2^8 x 3^4 x 5^2 x 7^1"
```

Python

```python
def prime_factorize(x):

  myprimes = list(primerange(2, x))
```

```
  factors = [i for i in myprimes if x%i==0]

  if len(factors) == 0:
    return str(x) + " is prime"

  my_factorization = pd.DataFrame(columns=['factor', 'power'])
  for i in factors:
    times = prime_n(x, i)
    new_row = pd.DataFrame([[i, times]], columns=['factor', 'power'])
    my_factorization = pd.concat([my_factorization, new_row], ignore_index=True)

  return format_factorization(my_factorization)
```

```
prime_factorize(34253) # should output "34253 is a prime number"
```

```
'34253 is prime'
```

```
prime_factorize(3628800) # should output "2^8 x 3^4 x 5^2 x 7^1"
```

```
'2^8 x 3^4 x 5^2 x 7^1'
```

1.9 Skill: User-proofing your function

It is never safe to assume that your user knows what they are doing. Can you make your function from the previous part more robust by testing the user input to ensure that it conforms to your expectations?

Planning

What assumptions does your previous answer make about parameters? Numeric x Single value x

What do you need to test to ensure those assumptions are met? test type of x and length of x if type is wrong, stop if length is >1 then work with only 1st entry in x

R

```
prime_factorize <- function(x) {
  stopifnot(is.numeric(x))
  if(length(x) > 1) {
    warning("x should be a single value. Using only the first entry in x")
    x <- x[1]
  }

  # Use prewritten function, but don't get x --
  #   get only primes less than x
  myprimes <- generate_primes(2, x-1)
  # Check whether primes are factors
  factors <- myprimes[x%%myprimes==0]

  # Test whether x is prime
  if (length(factors) == 0) {
    return(paste(x, "is a prime number"))
  }

  # This code only runs if x is not prime
  my_factorization <- data.frame()
  for(i in factors) {
    times <- prime_n(x, i)
    my_factorization <- rbind(my_factorization,
                              data.frame(factor = i,
                                         power = times))
```

```
  }

  # Return the formatted factorization
  return(format_factorization(my_factorization))
}
```

```
prime_factorize("string")
```

```
Error in prime_factorize("string"): is.numeric(x) is not TRUE
```

```
prime_factorize(2:5)
```

```
Warning in prime_factorize(2:5): x should be a single value. Using only the
first entry in x

[1] "2 is a prime number"
```

Python

```
def prime_factorize(x):
  try:
    x = int(x) # convert to integer
  except ValueError as e:
    print(f"Error: {e}. x must be coercible to an integer")
  except TypeError as e:
    print(f"Error: {e}")
  else:
    myprimes = list(primerange(2, x))
    factors = [i for i in myprimes if x%i==0]

    if len(factors) == 0:
      return str(x) + " is prime"

    my_factorization = pd.DataFrame(columns=['factor', 'power'])
    for i in factors:
      times = prime_n(x, i)
      new_row = pd.DataFrame([[i, times]], columns=['factor', 'power'])
      my_factorization = pd.concat([my_factorization, new_row], ignore_index=True)

    return format_factorization(my_factorization)
```

```
prime_factorize("string")
```

```
Error: invalid literal for int() with base 10: 'string'. x must be coercible to an integer
```

```
prime_factorize([2, 3, 4, 5])
```

```
Error: int() argument must be a string, a bytes-like object or a real number, not 'list'
```

2 Numerical Integration

This section will walk you through implementing numerical integration of a function.

2.1 Skill: Installing Packages

1. Write code to install the `SciPy` package in Python. It contains functions you will need for the remainder of this exam. If you choose to install the package via the terminal, place your code in the bash chunk; otherwise, place your code in the python chunk.

Python

```
pip install scipy
```

```
%pip install scipy
```

2.2 Skill: Loading packages

Load the packages you just installed in Python using the chunk below. If you did not manage to install the package, write the code you think you should use to load the packages.

Hint: Scipy should be typed as `scipy` when loading the package. You will only need the `norm` function in the scipy package, so you may want to be selective as to how you load the package.

Python

```
from scipy.stats import norm
```

Thinking Critically

There are multiple ways to load python packages. If I want to use the 'bar' function from the package 'foo', explain how you would load the package in order for the following code to be valid. If there is no valid way to load the package and use the function as listed, then state that.

1. `foo.bar()` I would load the package as `import foo` and then reference `foo.bar()` when calling the `bar` function.
2. `bar()` I would load the `bar` function from the `foo` package using `from foo import bar`, which would allow me to call the `bar` function directly.
3. `f.bar()` I would load the `foo` package as `import foo as f`, which aliases `foo` as `f`, allowing me to call the `bar` function as `f.bar()`.
4. `bar.foo()` I cannot think of a way to load the foo package and call the bar function using this syntax.

2.3 Skill: Using Prewritten Functions

The `dnorm(x, mean = 0, sd = 1)` function provides the value of the normal probability distribution density function in R. The `rv = scipy.stats.norm(x, location = 0, scale = 1)` function defines a normal random variable, and then `rv.pdf()` can be used to get the probability density function in Python.

In R and Python, define a sequence of `x = -3, ..., 3` that has length 100. Calculate `y = f(x)`, where `f` is the normal PDF.

R

```
x = seq(-3, 3, length.out = 100)
y = dnorm(x)
```

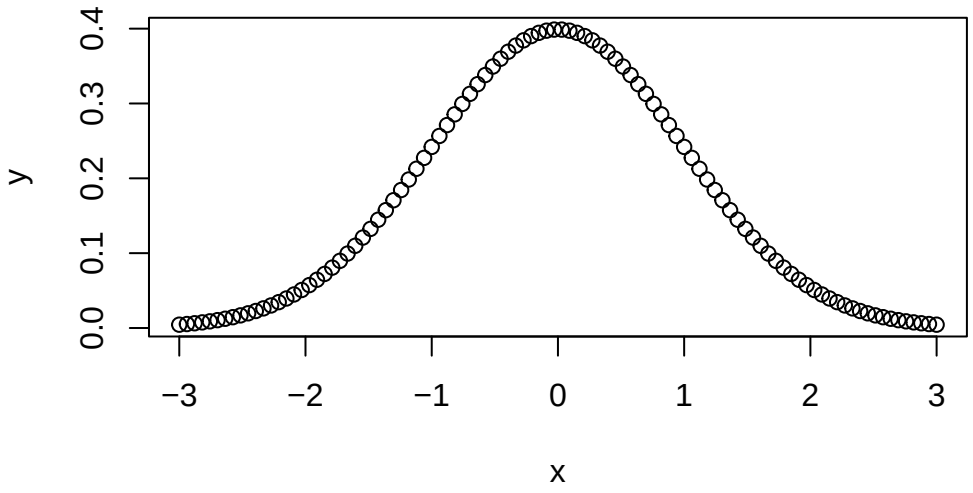
Python

```
import numpy as np
x = np.linspace(-3, 3, num = 100)
rv = norm()
y = rv.pdf(x)
```

Plot your data

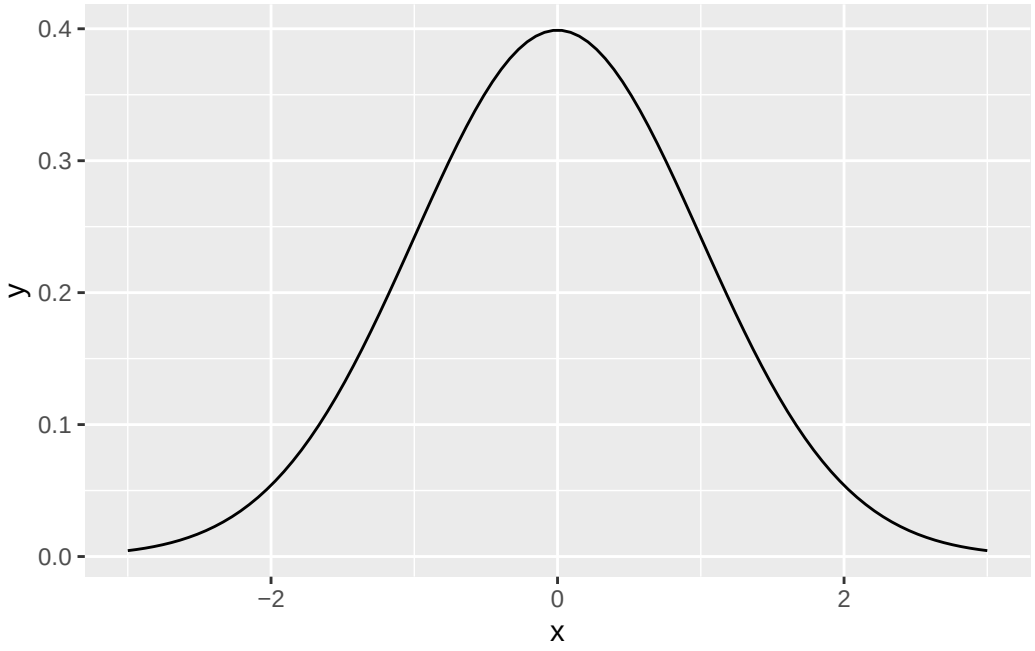
Create a line or scatter plot showing the relationship between x and $y = f(x)$.

```
plot(x, y, type = "p")
```

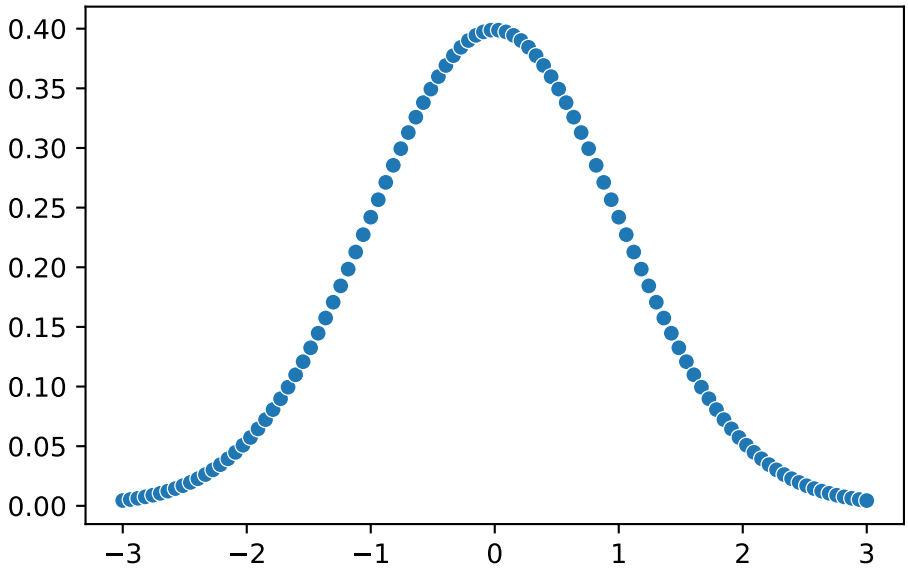



```
library(ggplot2)

ggplot() + geom_line(aes(x = x, y = y))
```




```
import seaborn as sns
import matplotlib.pyplot as plt
plot = sns.scatterplot(data = None, x = x, y = y)
plt.show()
```


0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
−3
−2
−1
0
1
2
3

2.4 Skill: Combining Sub Problems

Integrating the normal PDF to calculate $P(x \leq a)$ is extremely difficult to do by hand. For a long time before computers were common, statisticians used tables that contained the value of $P(x \leq a)$ for many different values of a. Luckily, with computers, we can avoid that!

The simplest method for numerical integration involves approximating $f(x)$ using a step function - essentially, evaluating $f(x)$ at evenly spaced x values, and then assuming that $f(x) \approx f(x_i)$ for $x \in [x_i, x_{i+1})$. Then, it is possible to turn $\int_a^b f(x)dx$ into $\sum_{x=x_0}^{x_n} f(x_i) \cdot (x_{i+1} - x_i)$

Write code to calculate the approximate integral of the normal distribution PDF from -3 to 3, using the x and y you created previously.

R

```
dx = (3 - (-3))/100 # this is actually somewhat approximate
integral = sum(y*dx)
integral
```

```
[1] 0.9875851
```

Python

```
dx = (3 - (-3))/100
integral = sum(y*dx)
integral
```

```
0.987585057855371
```

2.5 Skill: Writing Functions, Data Frames

Write a function, `norm_step` that takes arguments `a`, `b`, and `delta` and returns a data frame containing columns `x` and `y` that define the transition points (in `delta` increments) along the step function approximation to the normal pdf.

Hint:

- R: `seq(from, to, length.out = ...)` or `seq(from, to, by=...)`
- Python: `linspace(start, stop, num)` from the `numpy` package, or `arange(start, stop, step)` from the `numpy` package

Planning

Using the provided scratch paper (please put your name at the top), sketch a basic program flow map that shows how the code you've already written can be used to solve this problem. Identify any bits of logic you need to write to solve the problem.

My solution is sketched out on sheet _____

R

```r
norm_step <- function(a, b, step) {
  x <- seq(a, b, by = step)
  y <- dnorm(x)
  return(data.frame(x = x, y = y))
}
```

Python

```python
def norm_step(a, b, step):
  x = np.arange(a, b, step)
  rv = norm()
  y = rv.pdf(x)
  return pd.DataFrame({'x': x, 'y': y})
```

2.6 Skill: Functions, Mathematical Operations

Using your `norm_step` function, define a function `norm_int` that calculates the integral from `a` to `b` using step size `step`. Your function should call your `norm_step` function and should return a single numerical value.

R

```r
norm_int <- function(a, b, step) {
  df <- norm_step(a, b, step)
  return(sum(df$y*step))
}

norm_int(-4, 4, .01)
```

```
[1] 0.999938
```

Python

```python
def norm_int(a, b, step):
  df = norm_step(a, b, step)
  return sum(df.y*step)

norm_int(-4, 4, .01)
```

```
0.9999366485945341
```

2.7 Skill: Data Frames, Loops

Using your function, generate a table (data frame) showing $P(x < b)$ for each value of b between -2 and 2, at 0.1 increments. (e.g. $b = \{-2, -1.9, -1.8, ..., 0, 0.1, ..., 1.9, 2\}$)

You may assume that $a = -\infty$ or a value sufficiently negative that $P(x < a) \approx 0$ (-10 is probably good enough).

R

```r
# Define a sequence of b values
bval <- seq(-3, 3, .1)
res <- data.frame()

for(b in  bval) {
  res <- rbind(res, data.frame(b = b, pnorm = norm_int(-10, b, .01)))
}

head(res)
```

```
     b       pnorm
1 -3.0 0.001372168
2 -2.9 0.001895720
3 -2.8 0.002594892
4 -2.7 0.003519313
5 -2.6 0.004729397
6 -2.5 0.006297672
```

Python

```python
import numpy as np
import pandas as pd

# define a sequence of b values
bval = np.arange(-3, 3, .1)
res = pd.DataFrame()

for b in bval:
  new_row = pd.DataFrame([[b, norm_int(-10, b, .01)]], columns = ['b', 'pnorm'], dtype = ('f
```

```
  if res.shape[0] == 0:
    res = new_row
  else:
    res = pd.concat([res, new_row], ignore_index=True)

res.head()
```

```
     b     pnorm
0 -3.0  0.001328
1 -2.9  0.001836
2 -2.8  0.002516
3 -2.7  0.003415
4 -2.6  0.004594
```