Function Name: `stirling`

Inputs:

1. (*double*) An integer *n* greater than zero

Outputs:

1. *(double*) An approximation of *n* factorial

Background:

Oh no! In a contrived parody of a bad alien movie, aliens have invaded Earth and their advanced nanotechnological weapons are multiplying too quickly! Earth's top math/EE double majors soon realize that their numbers follow some sort of factorial function. However, you don't have time to do the regular way of finding the factorial, so the mathematicians provide you with an efficient formula to approximate the factorial: Stirling's formula! Now, go forth and use this amazing result from real analysis to put an end to the alien invasion.

Function Description:

Stirling's formula is a beautiful formula in mathematics that provides a scarily accurate approximation of the factorial:

$$n! \sim \frac{n^n}{e^n}\sqrt{2\pi n}$$

(In case you forgot, *n* factorial is defined as the product of all the integers from 1 to *n*.)

Write a function to take in an integer n, and use Stirling's formula to approximate the value of $n!$.

Notes:

- Round your output value to two decimal places.

Hints:

- The function `exp(n)` = e^n.
- `pi` is a built-in MATLAB constant.

Function Name: `strangerThings`

Inputs:

1. *(double)* Number of aliens
2. *(double)* Number of friends

Outputs:

1. *(double)* Number of aliens per friend
2. *(double)* Number of aliens left over

Background:

Your life has recently been turned upside down when demogorgons (aliens) infiltrated your small town. With psychic powers and baseball bats, you and your friends decide to go after the demogorgons yourselves. You lure the demogorgons to one location and wait in a nearby abandoned school bus. You and your friends decide to equally split the demogorgons between everyone there, and leave the leftover demogorgons for the local police. Since this is an extremely high-stress situation, you want to ensure you do your calculations right- so, you decide to write a MATLAB function to help you!

Function Description:

Given the number of aliens and the total number of friends, write a function that outputs how many aliens each person is responsible for fighting and how many aliens will be left over. All of your friends will be responsible for handling the same number of aliens. For example, if there are 5 aliens and 2 friends, each person would have to deal with 2 aliens, and 1 alien would be left over.

Example:

```
>> [aliensEach, aliensLeftover] = strangerThings(13, 3)
aliensEach => 4
aliensLeftover => 1
```

Hints:

- You may find the `floor()` and `mod()` functions useful.

Function Name: `spaceOddity`

Inputs:

1. (*double*) Semi-major axis of the other planet's orbit (in m)
2. (*double*) Mass of the other planet (in kg)

Outputs:

1. (*double*) Amount of time it takes for the other planet to go around the sun

Background:

You are Major Tom, and ground control isn't responding. (Typical of those lazy whippersnappers.) With nothing to do, you find yourself curious about how time would pass differently on Mars, and on other planets. Since a year on Earth is how long it takes the Earth to complete an orbit around the sun, how would this compare to a year on Mars, or how long it would take Mars to make a complete orbit around the sun?

Function Description:

Newton's law of gravitation gives the orbital period T of a body of mass M_1 around another body of mass M_2 with semi-major axis of orbit a as

$$T = 2\pi\sqrt{\frac{a^3}{G(M_1 + M_2)}}$$

where $G = 6.674 \times 10^{-11}$ m^3 kg^{-1} s^{-2} (the gravitational constant), a is in meters, T is in seconds, and M_1 and M_2 are both in kilograms.

Write a function that takes in the orbital axis and mass of a planet, and use the formula above to calculate how long it takes the other planet to go around the Sun once. The formula gives you an orbital period in seconds, so you should convert that time into Earth years.

Example:

Consider the planet Mars, with mass 6.39×10^{23} kg and semi-major axis of orbit 228×10^9 m. Using the formula and knowing the mass of the Sun (see the notes section), we have:

$$T = 2\pi\sqrt{\frac{(228 \times 10^9 \mathrm{m})^3}{6.674 \times 10^{-11}(6.39 \times 10^{23}\mathrm{kg} + 1.989 \times 10^{30}\mathrm{kg})}}$$

$$T = 5.94 \times 10^7 \mathrm{s}.$$

Converting to years, we get the final answer that Mars takes about 1.883 years to go around the Sun.

Continued...

Notes:

- The Sun's mass is 1.989×10^{30} kg. (Type it in MATLAB as `1.989e30`.)
- Round the output to three decimal places.
- For the gravitational constant *G*, use `6.674e-11` (type it exactly like that!).
- The semi-major axis of a planet's (elliptical) orbit is half the longest diameter of its orbit. (This isn't relevant to the problem, it's just a fun fact.)

Hints:

- All these worlds are ours, but there is one on which we should attempt no landing… try using the parameters of that world as the inputs to the solution function.

Function Name: `alienLetters`

Inputs:

1. (*char)* The first letter of the transmission, can be lower- or upper-case
2. (*char)* The second letter
3. (*char*) The third letter

Outputs:

1. (*double*) The ransom amount

Background:

After years of staring up into the stars with your telescope, your dream of making extraterrestrial contact has come true! An alien ship has appeared above the Clough Undergraduate Learning Commons and sent down an ominous message: they have abducted your friend! The aliens send a ransom note , but all is not what it seems. The message of the ransom note has only three letters, followed by instructions on how to use ASCII values to convert these letters into the desired amount of money to get your friend back, since they want to ensure your MATLAB prowess before the transaction. Time to get to work!

Function Description:

Write a function that takes in three letters, converts them all to lowercase, then finds the mean of the three letters. Finally, multiply this number by 1,000 and round to the nearest whole number.

Example:

```
>> out = alienLetters('L', 'o', 'L')
      out => 109000
```

Notes:

- Round your output to the nearest whole number.

Function Name: `clockHands`

Inputs:

1. *(double)* The current position of the hour hand
2. *(double)* The current position of the minute hand
3. *(double)* A positive or negative number of minutes

Outputs:

1. *(double)* The position of the hour hand after the specified time
2. *(double)* The position of the minute hand after the specified time

Background:

Some aliens stole your digital watch, so now you're back to the old school analog clocks! It is not always immediately obvious where the hands of a clock will be after a certain amount of time. It is even harder to visualize where the hands *were* some amount of time in the past. Luckily, this is not a very difficult problem for a computer to solve, so you will use that to your advantage.

Function Description:

This function will take in the current position of the hour hand, as an integer between `0` and `11` (`0` for noon/midnight), the current position of the minute hand, as an integer between `0` and `59` (`0` for "on-the-hour") and a positive or negative number of minutes elapsed.

Given this information, determine the new position of the clock hands. You should assume that the hour hand does not move until the next hour has begun. For example, the hour hand stays on `2` from 2:00 until 2:59 and only at 3:00 does the hour hand move to `3`.

Notes:

- The `mod()` and `floor()` functions will be useful.
- As you do this problem notice the behavior of `mod()` for negative inputs. This is a very important function in programming and will come up again and again in the class!

Hints:

- One way of solving this problem involves calculating the total number of minutes after noon/midnight before and after the given minutes have elapsed.
- Another way of solving it involves splitting the given number of minutes into a number of hours and a number of minutes.
- Pick whichever method makes more sense to you (or come up with your own method).