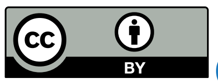
DT Challenge Python

Chatbot

1. Getting started with code
2. Data types: numbers and strings
3. Strings: working with words
4. Project 1
5. Making decisions
6. Investigating strings
7. Project 2
8. Repeating things
9. Project 3
10. Project 4: Putting it all together


(https://creativecommons.org/licenses/by/4.0/)

The Australian Digital Technologies Challenges is an initiative of, and funded by the Australian Government Department of Education and Training (https://www.education.gov.au/).

© Australian Government Department of Education and Training.

GETTING STARTED WITH CODE

1.1. Writing your first program

1.1.1. Hello, World!

Traditionally, the first program you write when learning a new programming language is Hello, World! (https://en.wikipedia.org/wiki/Hello_world_program) Let's write it in Python now:

```
print('Hello, World!')
```

```
Hello, World!
```

You can edit and run any example in Grok by clicking the ▶ button. Try changing `'Hello, World!'` to `'Hi!'`, and running it again.

Congratulations, your first Python program worked!

Click on the ↶ button to swap the code back to the original. Click it again to swap back to your version.

> **Our biggest hint!**
>
> Try running and modifying (messing around with even!) **every** example in these notes to make sure you understand it.

1.1.2. How to write programs

When you talk, you need to follow certain rules to be understood, called the *grammar* or *syntax* of the language.

You can't just use any words wherever you like. *like programming I* – doesn't make sense, but *I like programming* does!

Programming languages have syntax too. Python is pretty easy to learn because it has very simple syntax.

Have you noticed that the program is multi-coloured? The colourful *syntax highlighting* helps you code correctly:

```
print('Hello, World!')
```

```
Hello, World!
```

Purple tells you that `print` is a *function* and green tells you that `'Hello, World!'` is a *string*, which we'll talk about in a moment.

> **Watch the colours!**

Pay attention to these colours as you code. When they are not what you expect, there's often a typo that needs fixing.

1.1.3. What happens when things go wrong?

The Python *interpreter* is a program that reads and runs your code. Just like your English teacher, it will complain if you make spelling or grammar errors, e.g. *i can has cheezburger?* (https://en.wikipedia.org/wiki/I_Can_Has_Cheezburger%3F)

Unlike people, the interpreter can't understand bad grammar at all! Instead, it will stop with an error, e.g. `SyntaxError` or `NameError`.

Here we accidentally put `write` instead of `print`:

```
write('Hi There')
```

Python doesn't know they mean the same thing, so gives an error:

```
Traceback (most recent call last):
  File "program.py", line 1, in <module>
    write('Hi There')
NameError: name 'write' is not defined
```

Python displays the error (it does not recognise the name `write`) in red, including the type (`NameError`) and where it occurred (`line 1`).

Syntax highlighting saves the day!

Since `write` is not a builtin function, it didn't go purple like `print` does. The highlighting can help you catch syntax errors.

1.1.4. Help! I have a Syntax Error

If you get an error, don't panic, they happen all the time — we'll learn to fix them! Luckily lots of the errors you will make are easy to fix. Run the following example with an error in it:

```
print('Hello)
```

```
  File "program.py", line 1
    print('Hello)
                ^
SyntaxError: EOL while scanning string literal
```

We forgot to end the string `'Hello` with a quote. Python complains with a `SyntaxError` that it reached the *end of line* (EOL) without finding another quote. **Fix it by adding a quote right after** `Hello`.

Here's another broken program:

```
print 'Hello'
```

```
  File "program.py", line 1
    print 'Hello'
                ^
SyntaxError: Missing parentheses in call to 'print'
```

This time, we forgot the round brackets around what we wanted to `print`. Again, the interpreter say there is a `SyntaxError` on line 1.

Python can't always find the error

> The Python interpreter attempts to pinpoint the error, but it doesn't always get it right. If the highlighted bit doesn't seem like an error try looking earlier in the line or on the previous line.

1.1.5. Problem: Hello, World!

Write a program that prints out the message:

```
Hello, World!
```

If you're not sure how to start writing the program, go back a few pages and take another look at the notes.

How do I submit?

1. Write your program (in the `program.py` file) in the editor (large panel on the right);
2. Run your program by clicking Run in the top right-hand menu bar. The output will appear below your code. **Check the program works correctly!**
3. Mark your program by clicking Mark and we will automatically check if your program is correct, and if not, give you some hints to fix it up.

Testing

- ☐ Testing that the words are correct.
- ☐ Testing that the whitespace is correct.
- ☐ Testing that the punctuation is correct.
- ☐ Testing that the capitalisation is correct.
- ☐ **Hurrah, you got *everything* right!**

1.1.6. Problem: Therefore I rock!

You've printed your first text - congratulate yourself! Write a program that prints out:

```
Therefore, I rock!
```

Remember that the marker is really picky about punctuation and spelling.

Testing

- [ ] Testing that the words are correct.
- [ ] Testing that the whitespace is correct.
- [ ] Testing that the punctuation is correct.
- [ ] Testing that the capitalisation is correct.
- [ ] **Nice work! You got *everything* right!**

1.2. Python strings and variables

1.2.1. A string of characters

A *string* (the green messages) can contain any letters, digits, punctuation and spaces that you want, and it can be any length:

```
print('abc ABC 123 @!?.#')
```

```
abc ABC 123 @!?.#
```

The individual letters, digits, symbols and spaces are called *characters* and the word string is short for *string of characters*.

What if the character you want to use is a single quote? Python allows you to use double quotes around strings instead:

```
print("This message contains 'single quotes'.")
```

```
This message contains 'single quotes'.
```

`print` adds a newline

Notice that `print` also moves the output position (the *cursor*) to the next line (so the output ends on a blank line). Programmers call this printing a *newline* character after the string.

1.2.2. Using multiple print statements

A *statement* is the smallest stand-alone part of a program. It tells the computer to do something. Using the `print` function is an example of a statement.

Python will run statements in order (unless told otherwise), so to print multiple messages you can use:

```
print('Happy Birthday to you!')
print('Happy Birthday to you!')
print('Happy Birthday dear Python.')
print('Happy Birthday to you!')
```

```
Happy Birthday to you!
Happy Birthday to you!
Happy Birthday dear Python.
Happy Birthday to you!
```

Each message is printed on its own line. **Run the example to check!**

1.2.3. Storing things in variables

Writing out a long message many times is a pain. It would be great if we could just store the message somewhere and reuse it.

A *variable* is that place for storing a value so we can use it later.

Each variable has a *name* which we use to set and get its value. We create a new variable using an equals sign:

```
line = "For she's a jolly good fellow."
print(line)
print(line)
print(line)
print('And so say all of us!')
```

```
For she's a jolly good fellow.
For she's a jolly good fellow.
For she's a jolly good fellow.
And so say all of us!
```

The first statement creates a new variable called `line` (*a song line*) and stores the string `For she's a jolly good fellow.` We then `print` the value of the `line` variable three times.

> **Don't repeat yourself (DRY)**
>
> Avoiding repetition makes your code simpler, easier to read and edit, and so just more ... fun! Variables help avoid repetition.
>
> Imagine now we're singing to a male. Because we put the line in a variable, we only need to change `she` into `he` once!

1.2.4. Joining messages together

To join two or more strings together we can use addition, also called *concatenation*. For example, to print `'Harry'` followed by `'Potter'`, we can write:

```
print('Harry' + 'Potter')
```

which gives this (surprising?) output:

```
HarryPotter
```

They are added together exactly as they are, with no space! You need to add the space yourself if you want it:

```
print('Harry' + ' ' + 'Potter')
```

```
Harry Potter
```

We can also join strings that are stored in variables:

```
name = 'Vernon Dursley'
print(name + ' is a muggle!')
```

```
Vernon Dursley is a muggle!
```

Notice that we included a space in the `' is a muggle!'`.

1.2.5. Problem: I have no homework

Sometimes, if you repeat something often enough, it seems true. Repeating an opinion over and over again to make it seem true is called argumentum ad nauseam (http://rationalwiki.org/wiki/Argumentum_ad_nauseam), or *argument by repetition*.

We've written a program in the editor that repeats a statement.

Click ▶ Run to see what it does.

The statement is stored in a variable called `suggestion`.

Update this program so that it works for a different suggestion: `'I have no homework.'` Your updated program should print the message:

```
I have no homework.
I have no homework.
I have no homework.
I have no homework.
I have no homework.
No really! I have no homework.
```

Only change the `suggestion` variable!

You just need to change the value of `suggestion` to be `'I have no homework.'` instead of `'My room is clean.'`, run it to check it works, and then mark it.

You'll need

program.py

```
suggestion = 'My room is clean.'
print(suggestion)
print(suggestion)
print(suggestion)
print(suggestion)
print(suggestion)
print('No really! ' + suggestion)
```

Testing

- ☐ Testing that the words are correct.
- ☐ Testing that the whitespace is correct.
- ☐ Testing that the punctuation is correct.
- ☐ Testing that the capitalisation is correct.
- ☐ **Great work! Keep working on your Jedi mind tricks!**

1.2.6. Problem: Library Day

You need to remember to bring your library bag and book returns on Library day.

We've put a program in the editor that reminds you what day is Library day.

Click ▶ Run to see what it does.

The day you need to bring your book bag and returns is stored in a variable called `library_day`.

Disaster! Library day has changed from Friday to Wednesday so your reminder doesn't work.

Update this program so that it works for a different day, `Wednesday`. Your updated program should print the message:

```
Don't forget, Library Day is Wednesday!
On Wednesday you need to remember your library bag!
```

Only change the `library_day` variable!

You just need to change the value of `library_day` to be `Wednesday` instead of `Friday`, run it to check it works, and then mark it.

You'll need

program.py

```
library_day = 'Friday'
print("Don't forget, Library Day is " + library_day + '!')
print('On ' + library_day + ' you need to remember your library bag!')
```

Testing

- ☐ Testing that the words are correct.
- ☐ Testing that the whitespace is correct.
- ☐ Testing that the punctuation is correct.
- ☐ Testing that the capitalisation is correct.
- ☐ **Great work, we love borrowing books!**

1.3. Reading user input

1.3.1. Assigning to a variable

Setting the contents of a variable is called *assigning* a value to the variable. Python creates variables by assignment.

When you assign a new value to an existing variable, it replaces the old contents of the variable:

```
greeting = 'Hello!'
print(greeting)
greeting = 'Bonjour!'
print(greeting)
```

The old value in `greeting` (`'Hello'`) is replaced with the new value (`'Bonjour!'`) before the second `print` statement, producing:

```
Hello!
Bonjour!
```

Variables are like files

Variables are like files on your computer: they have a name and you can store data in them. You can look at the contents or overwrite the contents with new data.

1.3.2. Asking the user a question

Let's write a program that asks the user for information:

```
name = input('What is your name? ')
print(name)
```

Run this program. Even if you haven't run any so far, run this one! **You will need to type a name and press** `Enter`:

```
What is your name? Sandra
Sandra
```

The program prints the prompt `What is your name?` and waits for the user to type a name and press `Enter` (also called `Return`). The program then prints the name the user entered and stops.

A *prompt* is a message that tells (or prompts) the user that the program is asking for input (in this case, the user's name).

Run it again with a different name. Then try changing the prompt!

1.3.3. Calling functions

We have now used two Python functions, `print` and `input`, so we better tell you what a function actually is!

A function is a named* piece of code that performs a specific task.

Using that name, you can run the code (programmers say *call the function*) to perform the task without having to know how it works.

A function is called by name followed by round brackets.

Some functions take data to perform their task: `print` takes the value you want to print. This data goes inside the brackets:

```
print('To be, or not to be...')
```

```
To be, or not to be...
```

Some functions produce data while performing their task: `input` produces the string that the user entered. Programmers call this the *return* value. It can be used directly or stored in a variable:

```
msg = input('Repeat? ')
print(msg)
print(msg)
```

* some (anonymous) functions don't have a name!

1.3.4. Problem: Echo! Echo!

There is a game that little kids play when they're first learning to speak. It's called Echo!

It's a pretty easy game. You just need to repeat whatever was just said.

You are going to program this game. You need to get input from the user and print back exactly what the user input.

```
What do you want to say? Echo!
Echo!
```

Your program should work with anything the user types:

```
What do you want to say? I am having a great day!
I am having a great day!
```

Testing

- ☐ Testing that the words in the prompt are correct.
- ☐ Testing that the punctuation in the prompt is correct.
- ☐ Testing that the capitalisation in the prompt is correct.
- ☐ Testing that the white space in the prompt is correct.
- ☐ Testing with the word `Echo!`
- ☐ Testing with the word `I am having a great day!`
- ☐ Testing a hidden test case (to make sure your program works in general).

1.3.5. Variable variables!

Now we see why variables are called *variables*! When you run the program and ask the user for input, they could type anything:

```
animal = input('Favourite animal? ')
print('I like ' + animal + ' too!')
```

```
Favourite animal? tigers
I like tigers too!
```

Here, the `animal` variable contains `'tigers'`, but if the user types in something else, it will contain something else:

```
Favourite animal? pineapples
I like pineapples too!
```

This time, the `animal` variable contains `'pineapples'`.

Variables are *variable* because you may not know their value when you write the program, it could be *anything*!

Variables (or pronumerals) in algebra

In programming, variables store values that change or may be unknown before the program runs. In algebra, variables (like x and y) represent numbers that may vary or be unknown.

1.3.6. Problem: Meet the Puppy

Your friend has a new puppy! Write a program to tell the puppy to sit using its name.

```
What is the name of the puppy? Fluffy
Sit Fluffy
```

Your program should work with any puppy name:

```
What is the name of the puppy? Shadow
Sit Shadow
```

When you run your program, it should wait for you to type in the puppy's name, using `input`, then use the same name that the user typed in when printing the message.

> **Get the prompt string right!**
>
> Make sure you give `input` the same prompt message in the example above, especially **the space after the question mark**.

Testing

- [ ] Testing that the words in the prompt are correct.
- [ ] Testing that the punctuation in the prompt is correct.
- [ ] Testing that the capitalisation of the prompt is correct.
- [ ] Testing that the whitespace in the prompt is correct.
- [ ] Testing that the words in the sit command are correct.
- [ ] Testing that the punctuation in the sit command is correct.
- [ ] Testing that the capitalisation in the sit command is correct.
- [ ] Testing that the whitespace in the sit command is correct.
- [ ] **Yay, you've got the first example right!**
- [ ] Testing the second example in the question (when the user enters `Shadow`).
- [ ] Testing with the name `Fido`.
- [ ] Testing a two word name (`Shaggy Dog`).
- [ ] Testing a hidden test case (to make sure your program works in general).
- [ ] Testing another hidden test case.

1.3.7. Problem: Cheer me on!

Several of your friends are competing in a bike race and you want to cheer them all on! Write a program to write a cheer for you to shout for any friend's name.

```
Friend: Jane
Go Jane Go!
```

Your program should work with any friend's name:

```
Friend: Isabella
Go Isabella Go!
```

When you run your program, it should wait for you to type in the friend's name, using `input`, then use the same name that the user typed in when printing the message.

Get the prompt string right!

Make sure you give `input` the same prompt message in the example above, especially **the space after the colon**.

Testing

- ☐ Testing that the words in the prompt are correct.
- ☐ Testing that the punctuation in the prompt is correct.
- ☐ Testing that the capitalisation of the prompt is correct.
- ☐ Testing that the whitespace in the prompt is correct.
- ☐ Testing that the words in the cheer are correct.
- ☐ Testing that the punctuation in the cheer is correct.
- ☐ Testing that the capitalisation in the cheer is correct.
- ☐ Testing that the whitespace in the cheer is correct.
- ☐ **Yay, you've got the first example right!**
- ☐ Testing the second example in the question (when the user enters `Isabella`).
- ☐ Testing with the name `Vivek`.
- ☐ Testing a two word name (`Juan Vicente`).
- ☐ Testing a hidden test case (to make sure your program works in general).
- ☐ Testing another hidden test case.

1.4. Re-using variables

1.4.1. Reusable variables!

When you've stored some data in a variable, you can use it as many times as you want:

```
topic = input('What do you like? ')
print(topic + ' is so cool!')
print('I love ' + topic)
print(topic + ', ' + topic + ', ' + topic + '!')
```

```
What do you like? soccer
soccer is so cool!
I love soccer
soccer, soccer, soccer!
```

Try running it again with another topic (e.g., `Python`)!

A variable (such as `topic`) will keep the same value until the program finishes, unless you assign a new value to it.

1.4.2. Choosing good variable names

Choosing variable names is really important (it's an art!) **A good variable name clearly describes what the variable is and does** and this helps explain how the code works. Ideally, it is short too!

A bad name can make your program very confusing to read, e.g.:

```
blue = input('What colour is the sky? ')
print('The sky is ' + blue)
```

The name `blue` describes a possible value, not *what the variable represents* (the sky colour). What if the user enters something else?

```
What colour is the sky? pink
The sky is pink
```

Now the variable `blue` holds `'pink'`. How confusing! A much better variable name is `sky_colour`:

```
sky_colour = input('What colour is the sky? ')
print('The sky is ' + sky_colour)
```

It describes *exactly* what the variable represents, even if it is `pink`!

> **Variable name rules**
>
> Variable names cannot include spaces. Instead we used an *underscore* _ to join the words together into `sky_colour`.
>
> Variables also cannot contain punctuation or start with a digit.

1.4.3. Problem: Boaty McBoatFace

The UK's Natural Environment Research Council (NERC) asked the public to vote for the name of a new research ship (https://nameourship.nerc.ac.uk/).

The favourite (http://www.bbc.com/news/uk-england-36064659) was **Boaty McBoatface**, invented by James Hand.

Write a program to automate inventing names like Boaty:

```
Word? Boat
Boaty McBoatface
```

Here is another example:

```
Word? Truck
Trucky McTruckface
```

And the name of some code released by Google (https://research.googleblog.com/2016/05/announcing-syntaxnet-worlds-most.html):

```
Word? Parse
Parsey McParseface
```

In the end, the NERC went with RRS Sir David Attenborough (https://en.wikipedia.org/wiki/RRS_Sir_David_Attenborough), and named one of the ship's underwater vehicles *Boaty McBoatface*.

Testing

- ☐ Testing the words in the input prompt.
- ☐ Testing the punctuation, spaces and capitals in the input prompt.
- ☐ Testing the words in the first example from the question.
- ☐ Testing the punctuation, spaces and capitals in the first example.
- ☐ Testing the second example from the question.
- ☐ Testing the third example from the question.
- ☐ Testing with a different word (`Trunk`).
- ☐ Testing with `Grok`.
- ☐ Testing a hidden case.
- ☐ **Well done, Codey McCodeface!**

1.4.4. Problem: Pen Pineapple Apple Pen

Last year, fictional character DJ Piko-Taro caused an internet sensation with his very catchy tune PPAP (short for Pen Pineapple Apple Pen (https://www.youtube.com/watch?v=Ct6BUPvE2sM)).

This song is so simple and was so popular we think that a computer program could be used to come up with a similar song name.

Write a program to automate inventing songs like PPAP:

```
Type of fruit: apple
pen pineapple apple pen
```

Here is another example:

```
Type of fruit: cherry
pen pinecherry cherry pen
```

And one more. Did you know that the tomato can be classed as a fruit or a vegetable depending on who you're talking to (http://theplate.nationalgeographic.com/2015/02/09/fruit-or-vegetable/) - even in a court of law?

```
Type of fruit: tomato
pen pinetomato tomato pen
```

Testing

- ☐ Testing the words in the input prompt.
- ☐ Testing the punctuation in the input prompt.
- ☐ Testing the spaces and capitals in the input prompt.
- ☐ Testing the words in the first example from the question.
- ☐ Testing the punctuation, spaces and capitals in the first example.
- ☐ Testing the second example from the question.
- ☐ Testing the third example from the question.
- ☐ Testing with a different fruit (`banana`).
- ☐ Testing with a different fruit (`orange`).
- ☐ Testing a hidden case.
- ☐ **Well done! You'll be a meme sensation in no time!**

1.5. Variables

1.5.1. Using multiple variables

You can create as many variables as you need, as long as they have different names (otherwise, you're setting an existing variable).

```
first = input('What is your first name? ')
middle = input('What is your middle name? ')
last = input('What is your last name? ')
full_name = first + ' ' + middle + ' ' + last
print('Full name: ' + full_name)
```

The variable names make it very clear we've said a full name consists of a first, middle and last name. Self explanatory!

```
What is your first name? Hans
What is your middle name? Christian
What is your last name? Andersen
Full name: Hans Christian Andersen
```

1.5.2. Problem: Match of the Year!

It's the greatest match ever! Team 1 against Team 2... oh what's that? The team names are missing.

Write a program that asks for two team names, then prints out an announcement of the match.

Here is an example:

```
Who is team 1? Diamonds
Who is team 2? Silver Ferns
The match of the year: Diamonds vs. Silver Ferns!
```

Here's another example:

```
Who is team 1? Liverpool
Who is team 2? Everton
The match of the year: Liverpool vs. Everton!
```

Testing

- [ ] Testing the words in the first prompt message.
- [ ] Testing the words in the second prompt message.
- [ ] Testing the capitalisation of the prompts.
- [ ] Testing the punctuation in the prompts.
- [ ] Testing the spaces in the prompts.
- [ ] Testing the words in the match line.
- [ ] Testing the punctuation, spaces and capitals in the match line.
- [ ] **Testing the first example in the question.**
- [ ] Testing the second example in the question.
- [ ] Testing two rugby league teams.
- [ ] Testing two NBA basketball teams.
- [ ] Testing a hidden case.

1.5.3. Problem: Best New Ice Cream Combination

Everyone knows that two scoops of ice cream are better than one! But the best part is coming up with a new winning combination of flavours.

Write a program that asks for two ice cream flavours, then prints out an announcement of the new best ever flavour combination.

Here is an example:

```
Flavour 1: Chocolate
Flavour 2: Hazelnut
New best ever flavour combination...
Chocolate and Hazelnut!
```

Here's another example:

```
Flavour 1: Sour Cherry
Flavour 2: Cookies 'n' Cream
New best ever flavour combination...
Sour Cherry and Cookies 'n' Cream!
```

Testing

- ☐ Testing the words in the first prompt message.
- ☐ Testing the words in the second prompt message.
- ☐ Testing the capitalisation of the prompts.
- ☐ Testing the punctuation in the prompts.
- ☐ Testing the spaces in the prompts.
- ☐ Testing the words in the best ever flavour line.
- ☐ Testing the words in the ice cream combo line.
- ☐ Testing the punctuation, spaces and capitals in the ice cream combo line.
- ☐ **Testing the whole first example.**
- ☐ Testing the second example in the question.
- ☐ Testing with Mango and Chocolate.
- ☐ Testing with Green Tea and Black Sesame.
- ☐ Testing a hidden case.

1.6. Congratulations!

1.6.1. Congratulations!

Congratulations, you have completed the first module! You're starting to sound like a programmer!

You can now completely write and explain this code:

```
msg = input('Echo? ')
print(msg + '...' + msg + '...')
```

1. the call to `input` returns a string from the user;
2. that gets assigned to the `msg` (short for message) variable;
3. `msg` is concatenated with `'...'` twice;
4. and the call to `print` displays the result.

Keep going to module 2! Good luck!

DATA TYPES: NUMBERS AND STRINGS

2.1. Numbers

2.1.1. Python, the calculator

If there's one thing computers are really good at, it's working with numbers. They can do billions of calculations per second!

Speaking of seconds, let's calculate the number of seconds in a day:

60 seconds per minute × 60 minutes per hour × 24 hours per day

in Python, numbers are numbers, and multiply is `*` (*asterisk*):

```
print(60*60*24)
```

so the number of seconds in a day is:

```
86400
```

Like most programming languages, Python uses `*` for multiply because `x` is a variable name.

Try calculating the number of seconds in a year!

2.1.2. Python, the mega-calculator

Python is much better than a calculator, because it has variables!

You can use variables to store numbers and calculations for later, and describe what the calculation does very clearly:

```
secs_min = 60
mins_hour = 60
hours_day = 24
secs_day = secs_min*mins_hour*hours_day
print(secs_day)
```

```
86400
```

We've followed a consistent variable naming pattern or *convention*: seconds per minute (`secs_min`), minutes per hour (`mins_hour`), hours per day (`hours_day`), and seconds per day (`secs_day`).

This makes the variables and code easier to understand.

Magic numbers are bad

> Programmers hate to see strange numbers (like `86400`) in code. We call them *magic numbers* because it isn't clear what they do.
>
> Magic numbers should be put in variables that describe what they are, so they're not magical any more.

2.1.3. Mathematical operators

Python can do all the operations you expect from a calculator:

Name	Calculator	Python
add	+	`+`
subtract	-	`-`
multiply	×	`*`
divide	÷	`/`

Python uses `/` for division because ÷ isn't a key on most keyboards!

It uses the correct *order of operations*, so it will do multiply and divide before add and subtract, except if you use brackets. Let's calculate the hours in a leap year:

```
hours_leap_year = (365 + 1)*24
print(hours_leap_year)
```

```
8784
```

Run it, then try it again without the brackets. What happens?

2.1.4. Numbers and strings are different

Let's ask the user for two numbers and then add them together:

```
a = input('Enter a number: ')
b = input('Enter another number: ')
print(a + b)
```

```
Enter a number: 5
Enter another number: 6
56
```

Crazy! It doesn't do what we want at all – the answer should be `11`.

The reason is that **numbers and strings are different types of data**, and are treated differently in Python.

The problem here is that `input` always returns a *string* even if the user enters digits. Let's see that again without the `input` calls:

```
print('5' + '6')
```

```
56
```

Remember, adding two strings together joins them (*concatenation*).

So how do we convert the string from `input` into a number?

2.1.5. Converting strings to numbers

To treat a string as a number you must convert it first. The `int` function converts a string to an *integer* (a whole number):

```
print(int('5') + int('6'))
```

The `int('5')` call takes the string `'5'` and returns the integer `5`. The `int('6')` does the same for `'6'`. Now we're adding numbers:

```
11
```

So, now that we've discovered `int`, let's fix our original program:

```
a = input('Enter a number: ')
b = input('Enter another number: ')
print(int(a) + int(b))
```

If we always want to treat the value from `input` as a number, we can read the string and convert it to an integer in one go:

```
a = int(input('Enter a number: '))
b = int(input('Enter another number: '))
print(a + b)
```

When `5` is entered, `input` returns `'5'` (a string). This is given to `int`, which converts it to `5` (an integer). This is stored in variable `a`.

2.1.6. Problem: Twice as big

What's the biggest number you can think of? I bet I can think of one that's twice as big!

Write a program which asks the user for a number, and prints out a number twice as big as it.

We've given you a start with reading in the number.

Here's how your program should work:

```
What is the number? 1000
2000
```

Here's another example, with a small number:

```
What is the number? 2
4
```

Here's another example:

```
What is the number? 44444
88888
```

Converting the input into numbers

You'll need to convert the input into an integer using `int`. We've started you off with this line:

```
number = int(input('What is the number? '))
```

You'll need

program.py

```
number = input('What is the number? ')
number = int(number)
```

Testing

- ☐ Testing the words in the first example in the question.
- ☐ Testing the punctuation, spaces and capitals in the first example.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing a big number.
- ☐ Testing a small number.
- ☐ Testing a hidden case.
- ☐ **Great work!**

2.1.7. Problem: Next Olympics

The summer [Olympic Games (https://en.wikipedia.org/wiki/Olympic_Games)](https://en.wikipedia.org/wiki/Olympic_Games) happen every 4 years.

Write a program which asks for the year of the previous summer Olympics, then prints the year of the next Olympics (by adding 4).

Here's how it should work:

```
When is the Olympics? 2016
The next Olympics is in...
2020
```

Here's another example:

```
When is the Olympics? 2020
The next Olympics is in...
2024
```

Your program should still add 4, even if the year entered is wrong:

```
When is the Olympics? 2011
The next Olympics is in...
2015
```

> **Converting the inputs into numbers**
>
> You'll need to convert the input into an integer using `int`.

Testing

- [ ] Testing the words in the first example in the question.
- [ ] Testing the punctuation, spaces and capitals in the first example.
- [ ] Testing the second example in the question.
- [ ] Testing the third example in the question.
- [ ] Testing an Olympic year.
- [ ] Testing another Olympic year.
- [ ] Testing a hidden case.
- [ ] **Great work. Bring on Tokyo!**

2.2. More on strings

2.2.1. So why do strings need quotes?

Most of what you write in a Python program are instructions to follow, like `print()`. But text inside quotes is not treated as instructions. It is treated a sequence of letters to be used just as is (e.g. as a message). This is called a *string*.

This is how Python can tell the difference between printing the word `'message'` and printing the contents of the `message` variable.

```
message = 'hello'
print('message')
print(message)
```

```
message
hello
```

When there's no quotes, Python will try to interpret your words as instructions. Since the instructions don't make sense, Python will usually return an error.

```
Traceback (most recent call last):
  File "program.py", line 1, in <module>
    write('Hi There')
NameError: name 'write' is not defined
```

2.2.2. Single and double quotes

As mentioned before, quoted bits of text like `'Hello, World'` are called strings (short for string of characters). Strings are used to store any text data in a program (letters, words or sentences).

A string is a set of text characters enclosed by double (`"`) or single quotes (`'`). You need to end the string using the same quotation mark as you start it. For example these are okay (try running them):

```
print("hello")
```

```
hello
```

```
print('hello')
```

```
hello
```

However, the following example is incorrect because it starts with a double quote and ends with a single quote.

```
print("hello')
```

```
  File "program.py", line 1
    print("hello')
                 ^
SyntaxError: EOL while scanning string literal
```

2.2.3. Multiplying strings

You've already seen that we can do string addition:

```
print('ab' + 'ab')
```

```
abab
```

but something you might not expect is that we can also do string multiplication! Given that `'ab' + 'ab'` is `'abab'`, what would you expect `'ab' * 5` to be? Try it yourself:

```
print('ab' * 5)
```

```
ababababab
```

The `*` operator repeats a string a given number of times. It also works the other way around:

```
print(5 * 'ab')
```

```
ababababab
```

However, it only works with an integer and a string, it doesn't make sense to multiply two strings!

```
print('ab' * 'ab')
```

```
Traceback (most recent call last):
  File "program.py", line 1, in <module>
    print('ab' * 'ab')
TypeError: can't multiply sequence by non-int of type 'str'
```

2.2.4. Problem: Noooooooooo!

When writing a movie script, sometimes you want a simple “No” and sometimes a longer dramatic “Noooooooo”.

Write a program which asks how long the `No` should be, then uses that number of `o`'s. Here is an example:

```
How long? 1
No
```

Here is another example with more `o`'s:

```
How long? 10
Noooooooooo
```

Testing

- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing a length of 5.
- ☐ Testing a length of 4.
- ☐ Testing a length of 15.
- ☐ Testing a length of 0.
- ☐ Testing a hidden case.

2.2.5. Problem: Pull a happy face

Now that you're getting the hang of programming, you want to show everyone how happy you are! Write a program to help you generate Japanese-style emoticons (https://en.wikipedia.org/wiki/Japanese_Emoticons#Japanese_style) to express your joy. ^_^

Write a program which asks how happy you are, and prints out a face to match!

```
How happy? 1
^_^
```

Here is another example, where you're rather happy:

```
How happy? 3
^___^
```

Here is another example, where you're really happy:

```
How happy? 10
^__________^
```

Testing

- ☐ Testing the words in the input prompt.
- ☐ Testing the punctuation in the input prompt.
- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing a case when you are at happiness level 2.
- ☐ Testing a case when you are at happiness level 6.
- ☐ Testing a case when you are at happiness level 0.
- ☐ Testing a case when you are at happiness level 11.
- ☐ Testing a hidden case.

2.3. Mixing numbers and strings

2.3.1. Converting numbers to strings

You may have noticed that if you try to add a number and a string, you get an error:

```
answer = 5
print('The answer is ' + answer)
```

```
Traceback (most recent call last):
  File "program.py", line 2, in <module>
    print('The answer is ' + answer)
TypeError: Can't convert 'int' object to str implicitly
```

The `+` operator is confused here because we cannot add a string and an integer. When this happens, Python gives an error (in this case, a `TypeError` because strings and integers are different *types* of information).

To fix it we need to use the `str` function to turn the integer back to a string:

```
answer = 5
print('the answer is ' + str(answer))
```

```
the answer is 5
```

2.3.2. Working with numbers and strings

Now we can convert an input string into an integer using `int` and we can convert integers back to strings using `str`.

Using both `int` and `str`, we can write a program to read in numbers, do a calculation and print the result:

```
number = int(input("Enter a number: "))
answer = number + 10
print(str(number) + ' plus ten is: ' + str(answer))
```

```
Enter a number: 67
67 plus ten is: 77
```

Run this yourself with another number – then experiment with removing the calls to `int` or `str` and watch it break!

2.3.3. *F-strings*: Printing made easy

There's an easier way to print out strings and integers, and all types of information: formatted string literals, or "*f-strings*".

To use f-strings, we add the letter `f` **before** the start of string we want to use, and use curly braces `{` and `}` to use the values of variables in that string.

Here's an example:

```
dogname = input("What is your dog's name? ")
age = int(input('How old is your dog in years? '))
print(f'My dog {dogname} is {age} years old.')
```

```
What is your dog's name? Fluffy
How old is your dog in years? 6
My dog Fluffy is 6 years old.
```

Hint

Don't forget, you'll need to add an `f` to the string you want to print out *before* the opening quote! If you forget the `f`, this is what happens:

```
dogname = input("What is your dog's name? ")
age = int(input('How old is your dog in years? '))
print('My dog {dogname} is {age} years old.')
```

```
What is your dog's name? Fluffy
How old is your dog in years? 6
My dog {dogname} is {age} years old.
```

2.3.4. The power of F-strings

You can also modify the value that you want to print out from *inside* the format string:

```
dogname = input("What is your dog's name? ")
age = int(input('How old is your dog in years? '))
print(f'That makes {dogname} {age * 7} in dog years.')
```

```
What is your dog's name? Fluffy
How old is your dog in years? 6
That makes Fluffy 42 in dog years.
```

This doesn't change the value stored in the variable. It only changes what's printed out!

2.3.5. Problem: Letter from the Queen

For nearly 100 years, there has been a tradition that citizens of the United Kingdom and Commonwealth countries receive a letter from the King or Queen on their 100th birthday (https://en.wikipedia.org/wiki/Centenarian#British_and_Commonwealth_traditions).

Write a program that works out how long until your letter arrives!

Your program should ask the user how old they are then calculate 100 minus their age. For example:

```
How old are you? 14
You must wait 86 years.
```

Here is another example:

```
How old are you? 52
You must wait 48 years.
```

If you're already over 100, the answer is a negative number:

```
How old are you? 102
You must wait -2 years.
```

Testing

- ☐ Testing the first example from the question.
- ☐ Testing the second example from the question.
- ☐ Testing the third example from the question.
- ☐ Testing a very young person (3 years old).
- ☐ Testing someone who is almost there! (99 years old).
- ☐ Testing someone who is halfway there (50 years old).
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

2.3.6. Problem: Level Up!

Congratulations! Your character has levelled up!

Write a program that reads in what level the character was, and congratulates you on reaching the next level. For example:

```
Level: 2
Congratulations! You have reached level 3!
```

Here is another example:

```
Level: 15
Congratulations! You have reached level 16!
```

No matter what level you get to, you can always reach a level higher!

```
Level: 99
Congratulations! You have reached level 100!
```

Testing

- [ ] Testing the words of the prompt input.
- [ ] Testing the punctuation in the input prompt.
- [ ] Testing the spaces and capitals in the input prompt.
- [ ] Testing the whole first example.
- [ ] Testing the second example from the question.
- [ ] Testing the third example from the question.
- [ ] Testing with level 7.
- [ ] Testing with a very high level.
- [ ] Testing with an even higher level.
- [ ] Testing a hidden case.
- [ ] Testing another hidden case.

STRINGS: WORKING WITH WORDS

3.1. Uppercase and lowercase

3.1.1. Changing text to lowercase

When working with strings, we often want to change the case from uppercase to lowercase or vice versa. To do this, and many other string manipulations we can use *string methods*.

This is best demonstrated with an example.

```
msg = 'I know my ABC'
print(msg.lower())
```

```
i know my abc
```

The method we are using here is `lower`. The `msg` string contains a message in mixed case, and when you call the `lower` method it gives back a new message in lowercase only.

3.1.2. The original doesn't change

This can get a bit confusing, so look closely.

```
message = 'I know my ABC'
message.lower()
print(message)
```

```
I know my ABC
```

Even though we used the `lower` method, the string hasn't changed!

The `lower` method actually creates a new string which is the lowercase version of the original message. The original is not changed at all!

In this case we haven't saved the new lowercase version in a variable, so it's gone. Instead we should save it:

```
message = 'I know my ABC'
new_message = message.lower()
print(message)
print(new_message)
```

```
I know my ABC
i know my abc
```

Now we can use both the original and lowercase versions of the string.

3.1.3. Changing text to uppercase

Of course we might want to do the opposite of `lower`, and get the uppercase version of our string. The method for doing this is, not surprisingly, called `upper`:

```
msg = 'I know my ABC'
newmsg = msg.upper()
print(newmsg)
```

```
I KNOW MY ABC
```

Just like the `lower` method, `upper` is part of a family of methods that all return a new version of the string; the original string doesn't change.

Methods can also be used in expressions with a print statement, and only the output of the method will be printed. For example:

```
msg = 'I know my ABC'
print('Lowercase: ' + msg.lower())
print('Uppercase: ' + msg.upper())
print('Original: ' + msg)
```

```
Lowercase: i know my abc
Uppercase: I KNOW MY ABC
Original: I know my ABC
```

3.1.4. Problem: Shout it from the rooftops

Have you ever felt something so strongly you want to shout it from a rooftop?

Let's write a SHOUTER program that asks the user for some text and then SHOUTS it in upper case for us.

Here is an example interaction with the program:

```
Enter text: I love programming!
I LOVE PROGRAMMING!
```

Here is another example:

```
Enter text: It's lunchtime!
IT'S LUNCHTIME!
```

Testing

- ☐ Testing that the words in the prompt are correct.
- ☐ Testing that the punctuation in the prompt is correct.
- ☐ Testing that the capitalisation of the prompt is correct.
- ☐ Testing that the whitespace in the prompt is correct.
- ☐ Testing the words in the first example in the question.
- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing a short message.
- ☐ Testing a long message.
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

3.1.5. Problem: Breaking the fourth wall

An aside (https://en.wikipedia.org/wiki/Aside) is a dramatic device used to let a character speak directly to the audience, without the other characters hearing. It's a way of breaking the fourth wall (https://en.wikipedia.org/wiki/Fourth_wall). It is used in theatre, TV and film, and needs to be specifically marked as an *aside* in a script.

You're writing a play, and want to make sure your asides stand out. Write a program to help. Each aside should start with `Aside:` and be entirely in **lower case**.

Here is an example from Hamlet (https://en.wikipedia.org/wiki/Hamlet), by Shakespeare:

```
Line: A little more than kin, and less than kind.
Aside: a little more than kin, and less than kind.
```

Here is another example from The Emperor's New Groove (https://en.wikipedia.org/wiki/The_Emperor%27s_New_Groove):

```
Line: This is his story. WELL, ACTUALLY my story.
Aside: this is his story. well, actually my story.
```

Here's one more example from Ferris Bueller's Day Off (https://en.wikipedia.org/wiki/Ferris_Bueller%27s_Day_Off):

```
Line: How could I possibly be expected to handle school on a day like this?
Aside: how could i possibly be expected to handle school on a day like this?
```

Testing

- ☐ Testing the words in the first example.
- ☐ Testing the spaces in the first example.
- ☐ Testing the punctuation in the first example.
- ☐ Testing the capitalisation in the first example.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing a short aside.
- ☐ Testing a long aside.
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

3.2. Counting characters

3.2.1. Finding the length of a string

We can find out how long a string is using the `len` function:

```
print(len('Hello World!'))
```

```
12
```

Notice that the `len` function counts all the characters, including the 5 letters in `"Hello"`, the space, the 5 letters in `"World"`, and the exclamation mark.

As another example, we can count the number of letters in the alphabet, by first storing the letters of the alphabet in a variable `a`:

```
a = "abcdefghijklmnopqrstuvwxyz"
print(len(a))
```

```
26
```

3.2.2. Counting the characters in a string

Another useful string method is `count`, which allows you to count how many times a substring is contained in another string.

For example, to work out how many times `'l'` appears in `'hello world'`:

```
msg = 'hello world'
print(msg.count('l'))
```

```
3
```

This also works for multi-character strings. For example, to work out how many times double l appears:

```
msg = 'hello world'
print(msg.count('ll'))
```

```
1
```

Remember that the convention for calling string methods is that the string we are manipulating comes first, and then the method name, with any other information that is required passed in as arguments.

3.2.3. Problem: Twittier: Can I Tweet that?

Twitter (https://en.wikipedia.org/wiki/Twitter) is a social network where users post short "tweets" that used to be 140 characters or fewer. (Now they can be twice as long!) You've made your own version, "Twittier", that limits posts to only 44 characters!

Write a program to read in the post you'd like to make, and tells you how many characters you have spare.

```
Message: First post!
You have 33 character(s) left.
```

Here is another example:

```
Message: Can't wait until the weekend! #sleepin
You have 6 character(s) left.
```

Here's an example with a message of exactly 44 characters:

```
Message: This msg is exactly 44 characters - no more!
You have 0 character(s) left.
```

If the post is longer than 44 characters, it should print out the negative number of characters:

```
Message: Fitbits are like Tamagotchi, except you're trying to keep yourself alive.
You have -29 character(s) left.
```

Testing

- ☐ Testing the words in the first example.
- ☐ Testing the spaces in the first example.
- ☐ Testing the punctuation in the first example.
- ☐ Testing the capitalisation in the first example.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing the fourth example in the question.
- ☐ Testing another long post.
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

3.2.4. Problem: How good is your vocabulary?

If you're a fan of hangman you need to know some long words. English has a lot of very long words (https://en.wikipedia.org/wiki/Longest_word_in_English). The longest non-technical word in the English language is **antidisestablishmentarianism** (https://en.wikipedia.org/wiki/Antidisestablishmentarianism_(word)). At 28 letters it is a **very** long word.

You want to test your friends' vocabulary by seeing how close they can get to the longest word.

Write a program to read in the longest word your friends can think of and tell them how close they are to the length of "antidisestablishmentarianism":

```
Long word: incomprehensibilities
It is 7 letter(s) shorter than the longest word.
```

Here is another example:

```
Long word: unimaginatively
It is 13 letter(s) shorter than the longest word.
```

If the long word is longer than 28 characters, it should print out the negative number of letters

```
Long word: supercalifragilisticexpialidocious
It is -6 letter(s) shorter than the longest word.
```

But we all know that's not a *real* word.

Testing

- ☐ Testing the words in the first example.
- ☐ Testing the spaces in the first example.
- ☐ Testing the punctuation in the first example.
- ☐ Testing the capitalisation in the first example.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing another long post.
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

3.3. Parts of strings

3.3.1. Replacing parts of a string

You can replace part of a string (a substring) with another substring using the `replace` method. This requires you to pass the string data you wish to replace and the string you wish to replace it with as arguments to the method.

```
msg = 'hello world'
print(msg.replace('l', 'X'))
```

In this example, all the cases of the letter `'l'` are replaced by the letter `'X'`:

```
heXXo worXd
```

You can replace multiple characters at once:

```
msg = 'hello world'
print(msg.replace('hello', 'goodbye'))
```

```
goodbye world
```

And you can do replace multiple times in sequence:

```
msg = 'hello world'
msg = msg.replace('hello', 'goodbye')
msg = msg.replace('o', 'X')
print(msg)
```

```
gXXdbye wXrld
```

In this example, `'hello'` is replaced with `'goodbye'` and then `'o'` is replaced with `'X'`.

3.3.2. Problem: TELEGRAM STOP

Before telephones were invented, the only way to communicate quickly over long distances was by telegraph (https://en.wikipedia.org/wiki/Telegraphy). Short messages known as telegrams (https://en.wikipedia.org/wiki/The_Telegram) were sent over wires in Morse Code (https://en.wikipedia.org/wiki/Morse_code).

Telegrams were usually written all in uppercase letters, and instead of a full stop (`.`) they would write `STOP`.

Write a program which converts a message into a telegram by changing it to uppercase and replacing the full stops with the word `STOP` **including a space before it**. Here is an example:

```
Message: I will visit in April.
I WILL VISIT IN APRIL STOP
```

Telegrams would cost money per word, so they are usually short:

```
Message: French trip is awesome. See you soon.
FRENCH TRIP IS AWESOME STOP SEE YOU SOON STOP
```

Testing

- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing a message with no stops in it.
- ☐ Testing a message with many stops.
- ☐ Testing a message with extra stops.
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

3.3.3. Problem: Trackwork

Every so often, train tracks need maintenance or repairs. When that happens trains are replaced by buses. Write a program to help people know how to get where they're going.

Write a program which asks the user for their plans and replaces the word `train` with the word `bus`. Here is an example:

```
What are your plans? Catching the train to the zoo.
Catching the bus to the zoo.
```

Here is another example:

```
What are your plans? Take a train to Central then train to Redfern.
Take a bus to Central then bus to Redfern.
```

Sometimes there might be a funny outcome:

```
What are your plans? I am going to soccer training.
I am going to soccer busing.
```

Testing

- ☐ Testing the words in the first example.
- ☐ Testing the punctuation in the first example in the question.
- ☐ Testing the spaces and capitalisation of the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing a message with many trains.
- ☐ Testing a message with no trains.
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

PROJECT 1

4.1. Making simple games

4.1.1. Making simple games

Let's put what we've learnt so far into practice! In each of the Project modules in this course, we're going to write word games.

In this project, we'll start off with Mad Libs (https://en.wikipedia.org/wiki/Mad_Libs). Mad Libs is a game where one player asks the other for answers that fit a specific category, and then those answers are substituted in for blanks in a story.

We'll work up to more and more complex games in each project, and each project will start off small and build to something bigger.

4.1.2. Problem: Blank is the new blank

A common saying in fashion is to describe a new fashion colour or trend as "the new black". You can sometimes even see phrases like "x is the new y (https://snowclones.org/2007/07/01/x-is-the-new-y/)": Quiet is the new Loud (https://en.wikipedia.org/wiki/Quiet_Is_the_New_Loud), Bacon is the new Chocolate (https://www.theatlantic.com/magazine/archive/2005/11/better-bacon/304326/) or even Knitting is the new yoga (http://www.telegraph.co.uk/men/thinking-man/10552983/Mens-knitting-is-it-the-new-yoga.html).

Write a program that reads in two things, and prints out the resulting *new* phrase.

Here is an example:

```
x: Orange
y: Black
Orange is the new Black
```

```
x: Quiet
y: Loud
Quiet is the new Loud
```

Here's another example:

```
x: knitting
y: yoga
knitting is the new yoga
```

Testing

- ☐ Testing the words in the first prompt message.
- ☐ Testing the words in the second prompt message.
- ☐ Testing the capitalisation of the prompts.
- ☐ Testing the punctuation in the prompts.
- ☐ Testing the spaces in the prompts.
- ☐ Testing the words in the match line.
- ☐ Testing the punctuation, spaces and capitals in the match line.
- ☐ **Testing the first example in the question.**
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing another new comparison.
- ☐ Testing a hidden case.

4.1.3. Problem: Mad Libs 1: Letters from camp!

Mad Libs (https://en.wikipedia.org/wiki/Mad_Libs) are funny, often nonsensical stories built by asking someone for a series of words and using those words to fill in the blanks in a story. There are often crazy results!

Every time kids go away to camp, people expect them to send postcards. But there's more fun outside than being stuck inside writing! Write a program that will help your friends tell fun stories about what they're doing at camp. Your program should ask the user for some things you might find at camp and turn them into a story. For example:

Here is an example:

```
Name: Jane
Relation: Aunty
Noun: stick
Animal (plural): sloths
Sport: tennis
Adjective: bouncy
Another Adjective: crunchy
Verb: hop
Dear Aunty,
Camp stick has been bouncy so far!
Tomorrow we will play tennis if the weather is ok.
Today it has been raining cats and sloths all day!
If we cannot play tennis, maybe we will just hop.
I am sure it will be crunchy either way!
See you soon! Jane
```

Hint

Don't forget you can copy and paste the text for long passages to save on typing. You will need to use f strings to build the story. **Example:**

```python
relation = input('Relation: ')
print(f'Dear {relation},')
```

Testing

- [ ] Testing the words in the prompt messages.
- [ ] Testing the capitalisation in the prompt messages.
- [ ] Testing the punctuation in the prompts.
- [ ] Testing the spaces in the prompts.
- [ ] Testing the words and spaces in the first line of the letter.
- [ ] Testing the capitalisation and punctuation in the first line of the letter.
- [ ] Testing the words and spaces in the second line of the letter.
- [ ] Testing the capitalisation and punctuation in the second line of the letter.
- [ ] Testing the words and spaces in the third line of the letter.
- [ ] Testing the capitalisation and punctuation in the third line of the letter.
- [ ] Testing the words and spaces in the fourth line of the letter.

- ☐ Testing the capitalisation and punctuation in the fourth line of the letter.
- ☐ Testing the words and spaces in the fifth line of the letter.
- ☐ Testing the capitalisation and punctuation in the fifth line of the letter.
- ☐ Testing the words and spaces in the sixth line of the letter.
- ☐ Testing the capitalisation and punctuation in the sixth line of the letter.
- ☐ Testing the words and spaces in the seventh line of the letter.
- ☐ Testing the capitalisation and punctuation in the seventh line of the letter.
- ☐ **Testing the whole first example in the question.**
- ☐ Testing another example.
- ☐ Testing another example.
- ☐ Testing a hidden case.

4.1.4. Problem: Mad Libs 2: Bork Bork Bork!

Let's keep building up our [Mad Libs (https://en.wikipedia.org/wiki/Mad_Libs)](https://en.wikipedia.org/wiki/Mad_Libs)! The [Swedish Chef (https://en.wikipedia.org/wiki/Swedish_Chef)](https://en.wikipedia.org/wiki/Swedish_Chef) is a character from the Muppets who does hilarious things to food. His accent is quite thick, and he is often almost unintelligible.

Write a program to write out a three ingredients recipe in *Swedish Chef speak*. You should read in three ingredients from the user and replace all occurences of the letters `'th'` with the letter `'z'`.

For this question, Swedish Chef's recipes are always the same (with different ingredients), and they always ends in `'Bork! Bork! Bork!'`

Here is an example:

```
Ingredient 1: pastry
Ingredient 2: chicken thighs
Ingredient 3: beans
First cut ze pastry into triangles.
Then smash ze chicken zighs wiz a hammer.
Fry a mix of pastry and chicken zighs stirring gently.
Add in ze beans one ladle at a time.
Sprinkle ze remaining pastry over ze top.
Bork! Bork! Bork!
```

Here is another example:

```
Ingredient 1: onion
Ingredient 2: wheat thins
Ingredient 3: emmenthaler cheese
First cut ze onion into triangles.
Then smash ze wheat zins wiz a hammer.
Fry a mix of onion and wheat zins stirring gently.
Add in ze emmenzaler cheese one ladle at a time.
Sprinkle ze remaining onion over ze top.
Bork! Bork! Bork!
```

Hint

Don't forget you can copy and paste the text for long passages to save on typing.

Using f-strings to build the recipe will make things easier!

Testing

- ☐ Testing the words in the three prompt messages.
- ☐ Testing the capitalisation in the prompts.
- ☐ Testing the punctuation in the prompts.
- ☐ Testing the spaces in the prompts.
- ☐ Testing the words in the first line, **cutting into triangles**.
- ☐ Testing the words in the second line, **smashing with a hammer**.
- ☐ Testing the words in the third line, **frying the mix**.
- ☐ Testing the words in the fourth line, **adding ladles**.

- ☐ Testing the words in the fifth line, **sprinkling the garnish**.
- ☐ Testing the capitalisation, spaces and punctuation in the first line, **cutting into triangles**.
- ☐ Testing the capitalisation, spaces and punctuation in the second line, **smashing with a hammer**.
- ☐ Testing the capitalisation, spaces and punctuation in the third line, **frying the mix**.
- ☐ Testing the capitalisation, spaces and punctuation in the fourth line, **adding ladles**.
- ☐ Testing the capitalisation, spaces and punctuation in the fifth line, **sprinkling the garnish**.
- ☐ Testing the last line, `Bork! Bork! Bork!`
- ☐ **Testing the whole first example in the question.**
- ☐ Testing the second example in the question.
- ☐ Testing another example.
- ☐ Testing another example.
- ☐ Testing a hidden case.

MAKING DECISIONS

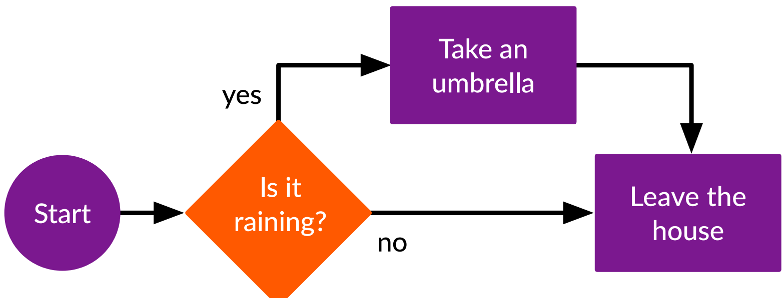
5.1. Making decisions

5.1.1. Why do we need decisions?

So far our programs have been a simple sequence of steps. The interpreter executes the statements from top to bottom, and so the program *runs the same way every time*.

In the real world, we **decide** to **take different steps** based on our situation. For example, if it's raining, we do an *extra step* of taking an umbrella before leaving the house.

This *flowchart* describes this process (or *algorithm*):



The diamond requires a `yes` or `no` decision. The answer determines which line we follow. If the answer is `yes`, we do the extra step of taking an umbrella. If the answer is no `no`, we skip it.

We can write this in Python using an `if` statement.

5.1.2. What if it is raining?

Let's write our flowchart as a Python program:

```
raining = input('Is it raining (yes/no)? ')
if raining == 'yes':
  print('Take an umbrella.')
print('Leave the house.')
```

Try it! What happens when you say `yes`, `no`, or any other answer?

Notice that the first `print` is *indented* (by two spaces). This is the *body* of the `if` statement. The body must be indented.

If the value stored in `raining` *is equal to* `'yes'` (because the user entered `yes`), then the body is run. Otherwise, it is skipped.

The second `print` always runs, because it is not indented, and isn't controlled by the `if` statement.

An `if` statement is a *control structure*

The `if` statement *controls* how the program runs by deciding if the body is run or not.

5.1.3. Controlling a block of code

An `if` statement can control more than one statement in its body.

These statements must have the same indentation, like this:

```
name = input('What is your name? ')
if name == 'Eliza':
  print('Nice name!')
  print('That is my name too!')
print('Pleased to meet you.')
```

If the name matches `'Eliza'` then all the indented lines (the *block*) will be executed first, then continue on with the rest of the program.

If it's anything else, those lines will be skipped and the next not-indented line will be executed next.

Careful with spaces!

The number of spaces of indent must be to the same depth for every statement in the block. This example is broken because the indentation of the two lines is different:

```
food = input('What food do you like? ')
if food == 'cake':
  print('Wow, I love cake too!')
    print('Did I tell you I like cake?')
```

You can fix it by making both `print` statements indented by the same number of spaces (usually 2 or 4).

5.1.4. Assignment vs. comparison

You will notice in our examples that we are using two equals signs to check whether the variable is equal to a particular value:

```
name = 'Amanda'
if name == 'Amanda':
  print('Hello, friend')
```

```
Hello, friend
```

This can be very confusing for beginner programmers.

A single = is used for *assignment*. This is what we do to set variables. The first line of the program above is setting the variable `name` to the value `"Amanda"` using a single equals sign.

A double == is used for *comparison*. This is what we do to check whether two things are equal. The second line of the program above is checking whether the variable `name` is equal to `"Amanda"` using a double equals sign.

If you do accidentally mix these up, Python will help by giving you a `SyntaxError`. For example, try running this program:

```
name = 'Amanda'
if name = 'Amanda':
  print('Hello, friend')
```

```
  File "program.py", line 2
    if name = 'Amanda':
            ^
SyntaxError: invalid syntax
```

Notice the second line only has one equals sign where it should have two.

5.1.5. Problem: Cheap Tuesdays

Your local cinema has a special on Tuesdays, so tickets are cheap! However, you want to hang out with your friends, so you are happy to go on any day.

Write a program to organise a day. If it's `Tuesday`, the program should print `Great! Tuesdays are cheap.` For example:

```
When can we see a movie? Tuesday
Great! Tuesdays are cheap.
I want to see Finding Dory.
```

If it's not `Tuesday`, you should still go to the movies:

```
When can we see a movie? Wednesday
I want to see Finding Dory.
```

Any answer other than `Tuesday` should work the same way:

```
When can we see a movie? Saturday
I want to see Finding Dory.
```

Testing

- ☐ Testing the words in the input prompt.
- ☐ Testing the capital, punctuation and spaces in the input prompt.
- ☐ Testing the first example in the question.
- ☐ Testing the punctuation, spaces and capital letters in the first example.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing a different day of the week (`Monday`).
- ☐ Testing a non-day name (`any day`).
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

5.1.6. Problem: Get in the game!

You're at a friend's board game party and it's time to play a new game. Your favourite game is [Codenames (https://en.wikipedia.org/wiki/Codenames_(board_game))](https://en.wikipedia.org/wiki/Codenames_(board_game)), so you're hoping to play that, but you're happy to play anything.

Write a program to see what the next game is. If it's `Codenames`, the program should print `I love Codenames!` For example:

```
What should we play next? Codenames
Awesome! I love Codenames.
I will set up the board!
```

Whatever the next game is, you're happy to play. Your program should print out that you will set up the board even if the next game isn't `Codenames`.

```
What should we play next? Settlers of Catan
I will set up the board!
```

Here is another example, with a different game:

```
What should we play next? Pictionary
I will set up the board!
```

Testing

- ☐ Testing the words in the input prompt.
- ☐ Testing the capital, punctuation and spaces in the input prompt.
- ☐ Testing the first example in the question.
- ☐ Testing the punctuation, spaces and capital letters in the first example.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing a different game (`Monopoly`).
- ☐ Testing a different game (`Race for the Galaxy`).
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

5.2. Decisions with two options

5.2.1. True or False?

`if` statements allow you to make *yes* or *no* decisions. In Python these are called `True` and `False`.

When you use an `if` statement Python works out if the condition is `True` or `False`. If the condition is `True` then the block controlled by the `if` statement will be run:

```
name = 'Sam'
if name == 'Sam':
  print('Hello, I think I know you?')
```

```
Hello, I think I know you?
```

If we change `name` the expression will evaluate to `False` and the block of code will be skipped:

```
name = 'Frodo'
if name == 'Sam':
  print('Hello, I think I know you?')
```

(note that when you run the program there is no output!)

Hint!

You can check whether the conditional expression is evaluating to `True` or `False` by testing it directly:

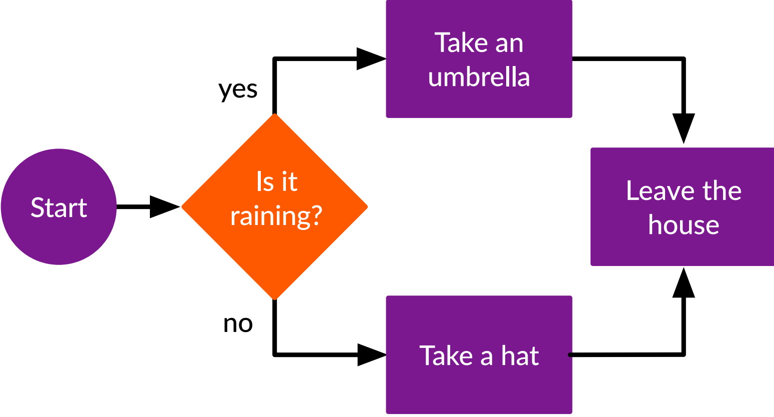
```
name = 'Sam'
print(name == 'Sam')
```

```
True
```

Try changing the name and seeing what happens.

5.2.2. Decisions with two options

All of the examples so far have done something when a condition is `True`, but nothing when the condition is `False`. In the real world we often want to do one thing when a condition is `True` and do something different when a condition is `False`.



If the user says `yes` it is raining, then the program should say to take an umbrella, but otherwise it should say to put on a hat.

What we want is an extra part to the `if` statement which is only run when the *condition* is `False`.

5.2.3. If it isn't raining...

In Python the `else` keyword specifies the steps to follow if the condition is `False` (in this case if it's **not** raining).

If the user says `yes` it is raining then the first *block* is executed, otherwise, the second *block* is executed instead:

```
raining = input('Is it raining? ')
if raining == 'yes':
  print('Pick up an umbrella.')
else:
  print('Put on a hat.')
```

Here, either the first or second `print` statement will be executed but not both. Notice the `else` keyword must be followed by a `:` character, just like the `if` statement.

5.2.4. Problem: Pikachu, I choose you!

Pokémon Go (https://en.wikipedia.org/wiki/Pok%C3%A9mon_Go) was downloaded over 100 million times in a month. We all have friends obsessed with Pokémon — and one of the favourites is Pikachu!

Write a program to talk to your Pikachu (https://en.wikipedia.org/wiki/Pikachu)-obsessed friend.

Your program should ask the user for their favourite Pokémon. If they say `Pikachu`, your program should print `Me too!` like this:

```
Which is your favourite? Pikachu
Me too!
```

If they say anything else, it should print `I like Pikachu.` instead:

```
Which is your favourite? Squirtle
I like Pikachu.
```

Testing

- ☐ Testing the words in the input prompt.
- ☐ Testing the capital, punctuation and spaces in the input prompt.
- ☐ Testing the words in first example in the question.
- ☐ Testing the punctuation, spaces and capital letters in the first example.
- ☐ Testing the words in the second example in the question.
- ☐ Testing the punctuation, spaces and capital letters in the second example.
- ☐ Testing with `Doduo`.
- ☐ Testing `Raichu`.
- ☐ Testing a typo.
- ☐ Testing a hidden case.

5.2.5. Problem: Protect Your Eyes

Did you know that blue coloured eyes have less melanin (https://en.wikipedia.org/wiki/Melanin#Effects) and are less protected from the sun than dark coloured eyes?

People with blue eyes should really put on some sunglasses as soon as possible! Write a program to warn them.

Your program should ask the user what colour their eyes are. If they say `blue`, your program should print `Slide on some sunnies pronto!` like this:

```
What colour are your eyes? blue
Slide on some sunnies pronto!
```

If they say anything else, it should print `Sunnies are cool but you can take your time.` instead:

```
What colour are your eyes? brown
Sunnies are cool but you can take your time.
```

If the answer is similar but not *exactly* `blue` you should still print out the other message. For example:

```
What colour are your eyes? bluey-green
Sunnies are cool but you can take your time.
```

Testing

- [ ] Testing the words in the input prompt.
- [ ] Testing the capital, punctuation and spaces in the input prompt.
- [ ] Testing the words in first example in the question.
- [ ] Testing the punctuation, spaces and capital letters in the first example.
- [ ] Testing the words in the second example in the question.
- [ ] Testing the punctuation, spaces and capital letters in the second example.
- [ ] Testing the third example in the question.
- [ ] Testing `grey` eyes.
- [ ] Testing a hidden case.
- [ ] Testing another hidden case.

5.3. Decisions about numbers

5.3.1. How do we compare things?

So far we have only checked whether two things are equal. However, there are other ways to compare things (particularly when we are using numbers). We can use the following *comparison operators* in `if` statements:

Operation	Operator
equal to	`==`
not equal to	`!=`
less than	`<`
less than or equal to	`<=`
greater than	`>`
greater than or equal to	`>=`

You can use a `print` statement to test these operators in conditional expressions:

```
x = 3
print(x < 10)
```

```
True
```

This prints `True` because 3 is less than 10.

```
x = 3
print(x > 10)
```

```
False
```

This prints `False` because 3 is not greater than 10.

5.3.2. Experimenting with comparison

Let's try some more examples to demonstrate how conditional operators work. Firstly, we have less than or equal to (`<=`):

```
x = 5
print(x <= 10)
```

```
True
```

Any value of `x` up to and including 10 will result in `True`. Any value of `x` greater than 10 will result in `False`. The opposite is true for greater than or equal to (`>=`).

Another important operator is not equal to (`!=`):

```
x = 5
print(x != 10)
```

```
True
```

Notice this program prints `True` because 5 is not equal to 10. This can be a bit confusing - see what happens if you change the value of `x` to 10.

5.3.3. Making decisions with numbers

Now we can bring together everything we've learned in this section, and write programs that make decisions based on numerical input. The example below makes two decisions based on the value in `x`:

```
x = 3
if x <= 3:
  print('x is less than or equal to three')
else:
  print('x is greater than three')
```

```
x is less than or equal to three
```

Try assigning different values to `x` to change how the program executes.

We could do something similar, but with user input:

```
height = int(input('How many cm tall are you? '))
if height == 157:
  print('You are the same height as Kylie Minogue.')
else:
  print('You are not the same height as Kylie Minogue.')
```

Remember you have to convert input to an integer using the `int` function if you want to do numerical comparisons.

5.3.4. Problem: Soil Sensor

You're creating an automatic system to water your plants. It has a sensor to measure the moisture in the soil. If the moisture level is below 25%, the plant should be watered.

Write a program to read in the moisture level from the sensor. If the moisture level is below 25%, it should do this:

```
Moisture level: 18
Water the plants.
```

If the moisture level is 25% or above, it should do this:

```
Moisture level: 27
The plants are happy.
```

Ideal soil moisture levels (https://en.wikipedia.org/wiki/Soil#Soil_moisture_content) are actually more complicated!

> **Hint: don't forget the `int`**
>
> Don't forget to convert the user's input from a string to an integer with `int` before doing the comparison.

Testing

- ☐ Testing the words in the input prompt.
- ☐ Testing the capital, punctuation and spaces in the input prompt.
- ☐ Testing the words in first example in the question.
- ☐ Testing the punctuation, spaces and capital letters in the first example.
- ☐ Testing the second example in the question.
- ☐ Testing that the program uses integers (with moisture level `100`).
- ☐ Testing a case where the plants are very dry.
- ☐ Testing the case with the minimum moisture level required to not water the plants.
- ☐ Testing the case where you only just need to water the plants.
- ☐ Testing a case where there is no moisture in the soil.
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

5.3.5. Problem: Buy me a pony

You want a pony! Horses and ponies are a little bit different. There is a height restriction. The International Federation for Equestrian Sports (https://en.wikipedia.org/wiki/International_Federation_for_Equestrian_Sports) defines a pony as no bigger than 149 cm from horse shoes to withers (https://en.wikipedia.org/wiki/Withers). If an animal's height is greater than 149 cm, the animal is a horse, not a pony.

Write a program to read in the animal's height. If the height is above 149 cm it should do this:

```
What height is the animal? 155
That animal is a horse.
```

If the height of the animal is 149 cm or lower, it should do this:

```
What height is the animal? 140
Buy me the pony!
```

The actual definition of a pony (https://en.wikipedia.org/wiki/Pony#Horses_and_ponies) is not quite so straight forward.

> **Hint: don't forget the `int`**
>
> Don't forget to convert the user's input from a string to an integer with `int` before doing the comparison.

Testing

- [ ] Testing the words in the input prompt.
- [ ] Testing the capital, punctuation and spaces in the input prompt.
- [ ] Testing the words in first example in the question.
- [ ] Testing the punctuation, spaces and capital letters in the first example.
- [ ] Testing the second example in the question.
- [ ] Testing that the program uses integers (with animal height `100`).
- [ ] Testing a case where the animal is very small.
- [ ] Testing the case with the maximum height for the animal to be a pony.
- [ ] Testing the case where the animal is just big enough to be a horse.
- [ ] Testing a case where the height is zero.
- [ ] Testing a hidden case.
- [ ] Testing another hidden case.

5.4. Making complex decisions

5.4.1. Decisions within decisions

The body of an `if` statement may contain another `if` statement. This is called *nesting*.

In the program below we have indented one `if` statement inside another one. That means the second condition will only be tested if the first condition is `True`.

```
x = 2
if x <= 3:
  print('x is less than or equal to three')
  if x >= 3:
    print('x is greater than or equal to three')
```

```
x is less than or equal to three
```

There is only one value of `x` that will cause both of these messages to be printed. Experiment to work out what it is.

5.4.2. Decisions with multiple options

If you want to consider each case separately, we need to test more conditions. One way of doing this would be using nesting, as shown below:

```
x = 5
if x < 3:
  print('x is less than three')
else:
  if x == 3:
    print('x is equal to three')
  else:
    print('x is greater than three')
```

```
x is greater than three
```

Another, neater, way of doing this is to use an `elif` clause. `elif` is an abbreviation for `else` and `if` together. It works like this:

```
x = 5
if x < 3:
  print('x is less than three')
elif x == 3:
  print('x is equal to three')
else:
  print('x is greater than three')
```

```
x is greater than three
```

You can test as many cases as you want by using multiple `elif`s.

5.4.3. Interplanetary visitor

We can now handle decisions with many options, like planets:

```
planet = input('What planet are you from? ')
if planet == 'Earth':
  print('Hello Earthling friend.')
elif planet == 'Mars':
  print('Hello Martian friend.')
elif planet == 'Jupiter':
  print('Hello Jovian friend.')
elif planet == 'Pluto':
  print('Pluto is not a planet!')
else:
  print('I do not know your planet.')
```

You could add as many `elif` clauses as you like, to deal with different cases.

5.4.4. Problem: Não compreendo?

You're in Brazil for the 2016 Summer Olympic Games! You're going to need a few Portuguese words to help you get around.

Write a program to translate some useful words from Portuguese (https://en.wikipedia.org/wiki/Portuguese_language). Here are the words you need:

Portuguese	English
Ola	Hello
Sim	Yes
Nao	No

Your program should ask for the Portuguese word then print out the English translation. For example:

```
What did they say? Ola
Hello
```

Here is another example:

```
What did they say? Sim
Yes
```

If the Portuguese word isn't one of the three above, your program should print `Nao compreendo.` which means “I don't understand.” in Portuguese. For example:

```
What did they say? Desculpe
Nao compreendo.
```

Testing

- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing the Portuguese word for `No`.
- ☐ Testing an different Portuguese phrase (`Por favor`, which means *please*).
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

5.4.5. Problem: Whose chair is whose?

The bears from the story Golidlocks and the Three Bears have very distinct property depending on their size.

The bears are Mama Bear, Papa Bear and Baby Bear. You want to be able to identify whose chair is whose based on the size.

Write a program to ask for the size of the chair and respond with whose chair it is. Here are the sizes and bears you will use:

Chair	Bear
big	That chair belongs to Papa Bear.
medium	That chair belongs to Mama Bear.
small	That chair belongs to Baby Bear.

Your program should ask for the size of the chair and then print out the chair's owner. For example:

```
What size chair? big
That chair belongs to Papa Bear.
```

Here is another example:

```
What size chair? small
That chair belongs to Baby Bear.
```

If the size isn't one of the three above, your program should print `'I wonder whose chair that is.'` For example:

```
What size chair? large
I wonder whose chair that is.
```

Testing

- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing a medium sized chair.
- ☐ Testing an different sized chair.
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

INVESTIGATING STRINGS

6.1. Investigating strings

6.1.1. Decisions about strings

So far we have written a lot of programs that take input from the user in the form of a *string* (a letter, word or sentence).

if statements to make decisions about strings like this:

```
food = input('Enter a food: ')
if food == 'mango':
  print('Mango is my favourite!')
else:
  print('Eww... ' + food)
```

In this example, we're checking if the food entered is *exactly equal* to `mango`. It doesn't match if the user enters `mangoes` or `Mango` with capital letters.

In this module we are going to learn how to manipulate these strings so that we can write more interesting programs that are a bit smarter.

6.1.2. Substrings within a string

Instead of checking if a string is exactly equal to what we expect we can also check for parts of a string (or *substrings*).

A substring could represent part of a word, or one word in a phrase. For example:

```
msg = 'concatenation is fun'
print('cat' in msg)
print('dog' in msg)
```

The first print statement will print `True` because the substring `'cat'` appears in the message. However, the second print statement will print `False` because `'dog'` does not appear.

```
True
False
```

Try changing the message and seeing what happens.

The `not in` operator does the opposite to `in`. It returns `True` when the string does *not* contain the substring:

```
msg = 'concatenation is fun'
print('cat' not in msg)
print('dog' not in msg)
```

```
False
True
```

6.1.3. Making decisions with strings

Like the other true/false expressions, we can now use these in an `if` statement to make decisions. For example to check if a person's name contains an `x` character you can write:

```
name = input('Enter your name? ')
if 'x' in name:
  print('Your name contains an x!')
else:
  print('No "x" in your name.')
```

Running this program gives:

```
Enter your name? Maxwell
Your name contains an x!
```

Notice that at the moment the program can only deal with lowercase letters (try entering `Xavier`). We'll solve this problem later in the section.

6.1.4. Problem: Forgotten Attachment?

Have you ever tried to send an email with an attachment but forgot to attach the file? Gmail and Outlook can check this for you.

Write a program that checks your email for the string **`attach`**, and if it's there, reminds you to attach the file. Here is an example:

```
Email: I will attach the document to this email.
Did you remember the attachment?
```

If **`attach`** doesn't appear in the email, then print that it was sent:

```
Email: Hi, how are you going?
Sent.
```

`attach` can appear *anywhere* in the email, even in a longer word:

```
Email: I have attached a photo.
Did you remember the attachment?
```

Testing

- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing another email with a missing attachment.
- ☐ Testing an email with no attachment.
- ☐ Testing a hidden case
- ☐ Testing another hidden case.

6.1.5. Problem: Moth spotting

Camouflaged moth (https://commons.wikimedia.org/wiki/File:Camouflaged_Moth_-_01.jpg)

There is a moth in that picture. Can you see it? Moths are very good at camouflage. They can even hide in text.

Write a program that checks a line of text for the string `moth`, and if it's there, announces that there is a moth hiding in the line. Here is an example:

```
Text: The moth circled the lamp.
I found a moth!
```

If `moth` doesn't appear in the line, then print that there are `"No moths here."`:

```
Text: Animals can be very good at hiding.
No moths here.
```

`moth` can appear *anywhere* in the line, even in a longer word:

```
Text: A good scout will always smother a campfire after use.
I found a moth!
```

Testing

- [ ] Testing the first example in the question.
- [ ] Testing the second example in the question.
- [ ] Testing the third example in the question.
- [ ] Testing another case of hiding moths.
- [ ] Testing a case with other winged insects.

- [ ] Testing a hidden case.
- [ ] Testing another hidden case.

6.1.6. When an `'apple'` is not an `'APPLE'`

Let's look at comparing strings that are lower and upper case.

To a computer, `'a'` and `'A'` are completely different *characters*, even if we interpret them as the same *letter*, just lower and upper case:

```
print('a' == 'A')
```

```
False
```

So testing if `'apple'` is the same as `'APPLE'` (or even `'Apple'`) will also be `False`:

```
print('apple' == 'APPLE')
```

```
False
```

We'll have to **modify** the strings so that we compare only lower case letters with lower case letters. (Or, only upper case with upper case!) We've seen how to change the case of a string using `lower` and `upper`:

```
word = 'Apple'
print(word.lower())
```

```
apple
```

6.1.7. Comparing strings but ignoring case

To compare whether strings are equal *ignoring case*, we have to modify the strings so that we compare only letters of the same case: lower with lower, and UPPER with UPPER. We can use `lower` and `upper` to make sure what we're comparing is the same.

In this example, we make new variables: `lower_word1` and `lower_word2`, and then compare them:

```
word1 = input('Word 1: ')
word2 = input('Word 2: ')
lower_word1 = word1.lower()
lower_word2 = word2.lower()
if lower_word1 == lower_word2:
  print('The words are the same, ignoring case.')
```

```
Word 1: banana
Word 2: BANANA
The words are the same, ignoring case.
```

We can actually skip making new variables, and just compare the `lower` versions of the two words:

```
word1 = input('Word 1: ')
word2 = input('Word 2: ')
if word1.lower() == word2.lower():
  print('The words are the same, ignoring case.')
```

6.1.8. A case study...

Here's an example to show how this can work in a problem. Let's write a program to ask the user their favourite movie.

If it's the same as yours (The LEGO Movie) let's say `Me too!` Otherwise, let's say `That is a good movie.` and tell them what our favourite is.

We want to accept answers like `The LEGO Movie`, `the lego movie` or even `the Lego MOVIE`.

```
movie = input('What is your favourite movie? ')
if movie.lower() == 'the lego movie':
  print('Me too!')
else:
  print('That is a good movie.')
  print('My favourite movie is The LEGO Movie.')
```

```
What is your favourite movie? the Lego movie
Me too!
```

Try out the example with a few different examples!

Comparing cases

Note that we're comparing whatever the user types in *turned into lower case* to something - in this case the string `'the lego movie'`.

If we're converting what the user types in into lower case, we'd better make sure we're comparing it to something in lower case!

We could do something like this:

```
if movie.lower() == 'The LEGO Movie'.lower():
```

That way we're making both sides lower case, so we can compare them directly.

6.1.9. Problem: Need a Hug?

Before there were emojis to help you express yourself in emails and texts, there was a text based system called emoticons (https://en.wikipedia.org/wiki/Emoticon#Japanese_style). In Western countries users generally had to tilt their head to the side to see the expression of the emoticon but in Japan they developed kaomoji (https://en.wikipedia.org/wiki/Emoticon#Japanese_style).

Write a program to give your friend a kaomoji hug if they need one. For example:

```
Do you need a hug? Yes
\(^-^)/
Have a great day!
```

It should work regardless of whether they used upper or lower case or a mix of both. For example:

```
Do you need a hug? YES
\(^-^)/
Have a great day!
```

If they don't need a hug you should still wish them a great day:

```
Do you need a hug? no
Have a great day!
```

Any answer other than `yes` (with any capitalisation) should work the same way:

```
Do you need a hug? nup
Have a great day!
```

Testing

- ☐ Testing the words in the input prompt.
- ☐ Testing the capital, punctuation and spaces in the input prompt.
- ☐ Testing the first example in the question.
- ☐ Testing the punctuation, spaces and capital letters in the first example.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing the fourth example in the question.
- ☐ Testing a case that isn't yes, but contains those letters.
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

6.1.10. Problem: Open Sesame!

Ali Baba and the Forty Thieves (https://en.wikipedia.org/wiki/Ali_Baba_and_the_Forty_Thieves) is a story that is often included in the collection of stories called the One Thousand and One Nights (https://en.wikipedia.org/wiki/One_Thousand_and_One_Nights). It is a folk tale that tells of a woodcutter who finds a secret cave that can only be opened with the voice command "Open Sesame".

Write a program to test is the password is correct to gain access to the cave. Remember that because it is a voice command it should work however the user types it. For example:

```
Voice command: open sesame
**cave opens**
Enter!
```

It should work regardless of whether they used upper or lower case or a mix of both. For example:

```
Voice command: Open Sesame
**cave opens**
Enter!
```

If the voice command is not right, your program should print:

```
Voice command: open please
**the cave does nothing**
```

Any answer other than `open sesame` (with any capitalisation) should work the same way:

```
Voice command: password
**the cave does nothing**
```

Testing

- ☐ Testing the words in the input prompt.
- ☐ Testing the capital, punctuation and spaces in the input prompt.
- ☐ Testing the first example in the question.
- ☐ Testing the punctuation, spaces and capital letters in the first example.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing the fourth example in the question.
- ☐ Testing a case that isn't quite right, but contains the right letters.
- ☐ Testing a case in all caps.
- ☐ Testing a case with strange capitalisation.
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

6.2. Characters in a string

6.2.1. Getting an individual character

Often we need to access individual characters in a string. Accessing a single character is done using the square bracket *subscripting* or *indexing* operation:

```
msg = 'hello world'
print(msg[0])
print(msg[1])
```

```
h
e
```

You might be wondering why we use `0` to access the *first* character in the string — why wouldn't we be using a `1`? This is because in programming, and in general in computer science, we start counting from 0 rather than from 1. So the first character in a string is at index `0`, the second character is at index `1`, the third character is at index `2`, and so on.

You can also access strings from the other end using a negative index:

```
msg = 'hello world'
print(msg[-1])
print(msg[-5])
```

```
d
w
```

6.2.2. Characters that don't exist

If you try and access a character past the end of the string, Python will throw an error and your program will crash. Because of this, it is important to make sure that you're always trying to access a character in the string which exists. Here is an example of what happens when you try to access a character which is past the end of the string:

```
msg = 'hello world'
print(msg[9])
print(msg[10])
print(msg[11])
```

```
l
d
Traceback (most recent call last):
  File "program.py", line 4, in <module>
    print(msg[11])
IndexError: string index out of range
```

The string `'hello world'` only has 11 characters, and we are trying to access character 12 (remember we start counting from 0).

6.2.3. Problem: Seeing Double

Lots of words in English end with a double letter! Words like `yell`, `too`, `baseball`, `happiness`, `free`, and so on.

Write a program that checks if the **last letter** and the **second last letter** of a word are the same character. Here's an example:

```
Word: yell
Ends with a double letter!
```

If the last two letters are different, your program should do this:

```
Word: something
No double letter at the end.
```

Double letters don't matter unless they're the last two, like this:

```
Word: yellow
No double letter at the end.
```

Your program will only be tested on words at least 2 letters long.

Testing

- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing the third example from the question.
- ☐ Testing a word with a double `oo` at the end.
- ☐ Testing a word with a double `ee` at the end.
- ☐ Testing a word with a double `ss` at the end.
- ☐ Testing a word with a double `nn` at the end.
- ☐ Testing a word with no double letter at the end.
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.
- ☐ Testing yet another hidden case.

6.2.4. Problem: Email Address

You're creating an email address for all the new students at your school. Each email address is created using the first letter of the student's first name and then their last name. Like this:

```
First name: Stephen
Last name: Merity
Your email address is smerity@example.school.edu
```

All of the letters in the email address should be **lowercase**.

Here is another example student at Example School:

```
First name: Jasmine
Last name: Constable
Your email address is jconstable@example.school.edu
```

Hint

This question has lots of moving parts! Start by breaking it up into smaller tasks and solve it bit by bit.

Testing

- ☐ Testing the letters in the first example in the question.
- ☐ Testing the case of the first example in the question.
- ☐ Testing the second example from the question.
- ☐ Testing a magical email address.
- ☐ Testing a vampiric email address.
- ☐ Testing a hairy email address.
- ☐ Testing a hidden case.

6.3. Checking string case

6.3.1. Testing the case of a string

We already learnt how to change the case of a string using `upper` and `lower`, but how do we know what case something is? We might need to check whether a user has entered upper or lowercase characters (for example, if they were entering a password.)

We can do this using the `isupper` and `islower` methods. These are similar to conditional operators - they return `True` or `False` depending on the input.

Here's an example that checks a lowercase string:

```
msg = 'a lowercase string!'
print(msg.islower())
print(msg.isupper())
```

```
True
False
```

Likewise, we can check an uppercase string, and get the opposite result:

```
msg = 'AN UPPERCASE STRING!'
print(msg.islower())
print(msg.isupper())
```

```
False
True
```

Change the `msg` variable to test what happens if you have a `'MiXeD CaSe'` string.

6.3.2. Making decisions about case

Just like before, we can now use these in an `if` statement to make decisions. For example to check if a user has given their name in the correct case, you could write:

```
name = input('Enter your name: ')
if name.isupper():
  print('Your name is in uppercase.')
elif name.islower():
  print('Your name is in lowercase.')
else:
  print('Your name is in mixed case.')
```

If we wanted to correct the user's input so that their name had traditional English capitalisation (the first letter in each word capitalised), we could use the `title` method:

```
name = input('Enter your name: ')
print('Name converted to:', name.title())
```

```
Enter your name: XAVIER
Name converted to: Xavier
```

Notice in all these examples, that punctuation is ignored.

6.3.3. Problem: Capital Cities

In English, city names have capital letters, but sometimes you forget to type the capital letters!

Write a program to correct this mistake. If the user enters a city name which is all lowercase, your program should capitalise it with `title`. Here is an example:

```
City name: auckland
You forgot the capital! It should be: Auckland
```

If the name is not all lowercase, then we assume that it's correct:

```
City name: San Francisco
Looks fine to me.
```

Here is one more example:

```
City name: brisbane
You forgot the capital! It should be: Brisbane
```

Testing

- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing a correctly capitalised city name.
- ☐ Testing a lowercase city name.
- ☐ Testing a city name with very strange capitals.
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

6.4. Slices of strings

6.4.1. One letter at a time

We have seen how to grab out a specific letter from a string:

```
msg = 'hello world'
print(msg[0])
print(msg[1])
```

```
h
e
```

This is called *indexing* and the number we use to look up each letter in the string is called the *index*.

In particular you should notice again that the first letter is index 0, the second letter is index 1 and so on.

You can also access strings from the other end using a negative index:

```
msg = 'hello world'
print(msg[-1])
print(msg[-5])
```

```
d
w
```

6.4.2. Substrings

If we wanted to create a new string which is the first 3 letters of another string, we could write this:

```
sentence = 'The cat is asleep.'
print(sentence[0] + sentence[1] + sentence[2])
```

```
The
```

But there's a nicer way!

A piece of a string (called a *substring*) can be accessed by using two numbers separated by a colon:

```
sentence = 'The cat is asleep.'
print(sentence[0:3])
```

```
The
```

This is called taking a *slice*.

A slice starts from the first number (index) and includes characters **up to but not including** the second index. So in the example below, looking at the string `x = 'hello world'`, slice `x[0:5]` starts from the `x[0]` and goes up to `x[5]` (the space) but doesn't include it. Here are a few more examples:

```
x = 'hello world'
print(x[0:5])
print(x[6:11])
```

```
hello
world
```

6.4.3. Substring to the end

If we want to find the *substring* until the end of the string, we can do that with a slice if we know the length of the string. We can find out the length of the string using the `len` function:

```
x = 'hello world'
print(x[3:len(x)])
print(x[6:len(x)])
```

```
lo world
world
```

But there's another way, if we leave out the second index in a slice (keeping the `:`!) then it will give us the substring up until the end of the string, just like it did with `len`:

```
x = 'hello world'
print(x[3:])
print(x[6:])
```

```
lo world
world
```

But what happens if we try to find a substring that doesn't exist?

```
x = 'hello world'
print('The substring is "' + x[20:30] + '"')
```

```
The substring is ""
```

The string `x` is less than 20 characters long so the substring `x[20:30]`, doesn't exist. Instead we get an empty string and it prints nothing.

6.4.4. Strings cannot be changed

Strings are *immutable*, which means that once the string is constructed, it cannot be changed. For example, you cannot change a character in an existing string by assigning a new character to a subscript:

```
x = 'hello world'
x[0] = 'X'
```

```
Traceback (most recent call last):
  File "program.py", line 2, in <module>
    x[0] = 'X'
TypeError: 'str' object does not support item assignment
```

You must instead create a new string by slicing up the string and joining together the pieces, and then saving the result back to the original variable:

```
x = 'hello world'
x = 'X' + x[1:]
print(x)
```

```
'Xello world'
```

Every operation that looks like it is modifying a string is actually creating a new string.

6.4.5. Problem: Sloth Speak

You're writing a book, with a character who is a sloth (https://en.wikipedia.org/wiki/Sloth). To make it clear in the book that your character speaks veeeery sloooowly you repeat letters in words that are said by this character.

Write a program to read in a single word and turn it into sloth speak. The word will always be in lowercase.

If the word **starts with a vowel**, you should repeat the **first** letter of the word 10 times:

```
Enter a word: and
aaaaaaaaaand
```

If the word **doesn't start with a vowel**, your program should repeat the **second** letter 10 times:

```
Enter a word: goodbye
goooooooooood bye
```

Here's another example:

```
Enter a word: hello
heeeeeeeeeello
```

Hint: Is this letter a vowel?

A nice way to check if a letter is a vowel is to check if it's one of the letters in the string `aeiou`.

```
letter = 'a'
if letter in 'aeiou':
  print('The letter is a vowel!')
```

Also remember that you can multiply strings by numbers using `*`.

Testing

- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing another case that starts with a consonant.
- ☐ Testing a case that starts with a vowel.
- ☐ Testing another case that starts with a vowel.
- ☐ Testing yet another case that starts with a vowel.
- ☐ Testing a cephalopod case.
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.
- ☐ Testing yet another hidden case.

6.4.6. Problem: Re-sounding Success

There is a common misconception that a [duck's quack doesn't echo (https://en.wikipedia.org/wiki/List_of_common_misconceptions#Biology)](https://en.wikipedia.org/wiki/List_of_common_misconceptions#Biology). Echoes are reverberations of sound and work best with long sounds like 'o' and 'ee'.

Write a program to read in a single word and turn it into echo speak. The word will always be in lowercase.

If the word **ends with an o**, you should repeat the **last** letter of the word 10 times:

```
Enter a word: echo
echooooooooooo
```

If the **second last and last letters are both e**, your program should repeat the **last** letter 10 times:

```
Enter a word: free
freeeeeeeeeee
```

Here's another example:

```
Enter a word: hello
hellooooooooooo
```

If the word nether ends with an o or a double e the program should print the original word.

```
Enter a word: goggles
goggles
```

> **Hint: Is this letter a vowel?**
> Remember that you can multiply strings by numbers using `*`.

Testing

- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing another case that ends with a consonant.
- ☐ Testing a case that ends with two ees.
- ☐ Testing a word that ends only in one e.
- ☐ Testing another case that ends with an o.
- ☐ Testing yet another case that ends with a vowel.
- ☐ Testing a flying case.
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.
- ☐ Testing yet another hidden case.

6.4.7. Checking the ends of a string

We've seen how to call a method on a string object (like `upper`), and introduced a couple of string methods for converting strings to uppercase and lowercase, and replacing bits of a string with other strings. Here are a few more string tricks...

Checking if a string starts or ends with a substring:

```
s = 'hello world'
print(s.startswith('hello'))
print(s.endswith('rld'))
```

```
True
True
```

6.4.8. Problem: Unbelieve-a-bull!

Rockhampton (https://en.wikipedia.org/wiki/Rockhampton) is known as the Beef Capital of Australia. Several businesses in the city have life-size promotional bulls with puns: lease-a-bull for a real estate agent, remove-a-bull for removalists, and train-a-bull for the local tafe.

Write a program that helps other businesses come up with puns for their bulls! Your program should read in a word. If the word ends with `'able'`, you should print out the punny version.

For example:

```
Word: fashionable
fashion-a-bull
```

If the word does not end with `'able'`, you should print out a different message:

```
Word: bovine
No bulls here
```

You should print out that message even if the word has `'able'` in it, but not at the end of the word. Here is another example:

```
Word: fabled
No bulls here
```

Here's one more example:

```
Word: deliverable
deliver-a-bull
```

What about tableable?

We'll only test this on words that have `'able'` in them *once*, so you don't have to worry about whether `'tableable'` should be come `'table-a-bull'` or `'t-a-bull-a-bull'`. That's up to you!

Testing

- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing the fourth example in the question.
- ☐ Testing an example for a drawing company: *illustrat-a-bull*.
- ☐ Testing an example for a washing company: *wash-a-bull*.
- ☐ Testing an example for a school that doesn't fit the pattern.
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

6.4.9. Problem: That's exciting!

Terry Pratchett, the author of the Discworld Series of books (https://en.wikipedia.org/wiki/Discworld), once wrote "Five exclamation marks, the sure sign of an insane mind." Nevertheless, you believe that your friends just aren't excited enough when writing good news.

Write a program that fixes your friends' sentences to be more excited (but not quite insane). When they type a sentence, if it ends with an exclamation mark your program should rewrite it in all caps and with 3 extra exclamation marks tacked on the end.

For example:

```
Line: That is exciting!
THAT IS EXCITING!!!!
```

If the word does not end with `'!'`, you should print out a different message:

```
Line: I like homework
That is not an exciting message.
```

A message is only exciting if it ends with `'!'`, so if it appears anywhere else, you should treat it like an unexciting message. Here is an example:

```
Line: But, soft! what light through yonder window breaks?
That is not an exciting message.
```

Here's one more example:

```
Line: Happy Birthday!
HAPPY BIRTHDAY!!!!
```

> **What about already excited sentences?**
> If the user types multiple exclamation marks we still add 3 to the end. If that means they are considered insane it's not our problem.

Testing

- [ ] Testing the first example in the question.
- [ ] Testing the second example in the question.
- [ ] Testing the third example in the question.
- [ ] Testing the fourth example in the question.
- [ ] Testing an example for a warning: *LOOK OUT!!!!*
- [ ] Testing an example for a cheer: *HIP HIP HOORAY!!!!*
- [ ] Testing an example that doesn't fit the pattern.
- [ ] Testing a hidden case.
- [ ] Testing another hidden case.

6.5. Comments

6.5.1. Commenting on your code

As you write longer and more complex programs, it can be very helpful to add notes to the code to remind you what it does.

In Python you can put a hash (#) on any line, and anything that comes after it will be ignored.

```
# This is just a comment, it doesn't do anything.
print('Hello')
# Another comment
```

```
Hello
```

This is useful to help explain a difficult part of the code, or to divide up code into steps.

6.6. Congratulations!

6.6.1. Congratulations!

Congratulations, you have just completed the module on Investigating strings!

In this module we learned a lot about strings and we covered the following topics:

- Finding if a character or string is inside another string;
- checking if strings are upper or lowercase;
- converting strings to all-caps or lowercase letters;
- replacing parts of a string;
- picking out characters from a string by number.

PROJECT 2

7.1. More word games

7.1.1. Making simple games

Let's put what we've learnt about strings and decisions into practice making some more complex games.

We're going to add in the concept of decisions and string manipulations. These will help us build up to more complex games as we keep adding more skills to our toolbox.

7.1.2. Problem: Taboo!

[Taboo (https://en.wikipedia.org/wiki/Taboo_(game))](https://en.wikipedia.org/wiki/Taboo_(game)) is a word game where one person describes a word on a card so that their partner can guess it. However, there are certain words that the person can't say when trying to describe the thing! A player might have to describe *'cereal'* without using the word *'breakfast'*.

Write a program to help play Taboo. Here's an example for trying to guess the word *'cereal'*:

```
Taboo word: breakfast
Description: A type of grain, like oats or bran.
Safe!
```

Here's another example, which includes the taboo word:

```
Taboo word: breakfast
Description: The thing you eat for breakfast.
Taboo!
```

You should print `Taboo!` even if the word is in another word. Here's an example when trying to guess the word *'wheel'*:

```
Taboo word: bike
Description: There's 4 on a car and 2 on a motorbike.
Taboo!
```

Sometimes games get exciting and players tend to shout. If the word occurs in **any case**, lower, upper or mixed, it still counts. For example, guessing the word *'oval'*:

```
Taboo word: sport
Description: WE HAVE SPORTS HERE!
Taboo!
```

Testing

- ☐ Testing that the words in the prompts are correct.
- ☐ Testing that the punctuation in the prompts is correct.
- ☐ Testing that the capitalisation in the prompts is correct.
- ☐ Testing that the white space in the prompt is correct.
- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing the fourth example in the question.
- ☐ Testing a game trying to guess *'water'*.
- ☐ Testing another game trying to guess *'water'*.
- ☐ Testing a tricky case.
- ☐ Testing a hidden test case (to make sure your program works in general).
- ☐ Testing another hidden case.

- [ ] Testing a tricky hidden case.
- [ ] Testing a final hidden case.

7.1.3. Problem: Word Chain!

Word chain (https://en.wikipedia.org/wiki/Word_chain) is word game where players take turns saying words that start with the last letter of the previous word. You might have played this game on long car trips.

Write a program to help you and your friends play word chain. Your program should read in two words and print out whether they are valid to follow each other. You should ignore case for this game.

Here is an example of a valid pair:

```
Word 1: carrot
Word 2: turnip
Valid
```

Here is another example of an invalid pair:

```
Word 1: orange
Word 2: apple
Nope!
```

The word `apple` is rejected because it doesn't start with the letter `e` from the previous word, `orange`.

Here's one more example, with uppercase and lowercase letters:

```
Word 1: Passionfruit
Word 2: Tangerine
Valid
```

Getting individual letters in a word

Take another look at this slide on string indexing (https://groklearning.com/learn/aca-dt-78-py-chatbot/stringdecisions/10/) for help on how to compare starting and ending letters!

Testing

- [ ] Testing that the words in the prompts are correct.
- [ ] Testing that the punctuation in the prompts is correct.
- [ ] Testing that the capitalisation in the prompts is correct.
- [ ] Testing that the white space in the prompt is correct.
- [ ] Testing the first example in the question.
- [ ] Testing the second example in the question.
- [ ] Testing the third example in the question.
- [ ] Testing a game with countries.
- [ ] Testing another game with countries.
- [ ] Testing a game with strange case.
- [ ] Testing another game with strange case.
- [ ] Testing a hidden test case (to make sure your program works in general).
- [ ] Testing another hidden case.

- [ ] Testing a final hidden case.

REPEATING THINGS

8.1. Looping with conditions

8.1.1. Introducing the `while` loop!

Using a `while` loop, we can repeat some code *while* a condition is true.

```
command = input('First command: ')
while command != 'stop':
  print('You entered ' + command)
  command = input('Command: ')
print('Stopped.')
```

Run it yourself and try it with several different inputs:

```
First command: something
You entered something
Command: something else
You entered something else
Command: help me
You entered help me
Command: stop
Stopped.
```

This program will keep repeating everything inside the loop *while* the last command entered was not **`stop`**.

If you never type in **`stop`** then the program will never finish!

8.1.2. Visualising a `while` loop

Let's take a closer look at what's going on here:

```
command = input('First command: ')
while command != 'stop':
  print('You entered ' + command)
  command = input('Command: ')
print('Stopped.')
```

The `while` keyword in Python asks a question, just like an `if` statement. The big difference is that it keeps repeating the code *inside* the loop until that question (or *condition*) isn't `True` anymore.

Ask user for the first command

Save that user input as command variable

Is command not equal to 'stop'?

True

Print the command

Ask user for the next command

Save that user input as command variable

False

Print 'Stopped.'

Following the arrows in this flowchart, look how the loop will only end when the *condition* is not true anymore. In this example it will only finish when `command != 'stop'` is `False`. So long as the condition is `True` the program will just keep on repeating forever!

8.1.3. Something is wrong?!

These loops can be pretty tricky to get right. Here are a few common mistakes when writing `while` loops:

Finished before you started!

In your code, before the `while` loop you have to set it up so that the *condition* is already `True` when it starts. Otherwise the whole `while` loop will be skipped. (Since it only runs while the condition is `True`!)

```
command = 'stop'
while command != 'stop':
  print('You entered ' + command)
  command = input('Command: ')
print('Stopped')
```

```
Stopped
```

Here the loop doesn't even start asking for input, because the `while` loop condition was already `False`!

Not setting up beforehand

In this example, the `while` loop is trying to ask the question 'Is the command variable not equal to stop?' but since the command variable hasn't been created yet, it doesn't know what to do!

```
while command != 'stop':
  print('You entered ' + command)
  command = input('Command: ')
print('Stopped')
```

```
Traceback (most recent call last):
  File "program.py", line 1, in <module>
    while command != 'stop':
NameError: name 'command' is not defined
```

See how it says "`name 'command' is not defined`"? That means you're trying to use the `command` variable but it hasn't been created (defined) yet.

8.1.4. My program won't stop!

Another common mistake with `while` loops is making an **infinite loop**.

Try running this example.

```
command = input('First command: ')
while command != 'stop':
  print('You entered ' + command)
print('Stopped.')
```

See how it just keeps going and going and going?

The `while` loop will keep repeating *while* the condition is `True`. In this example, we ask the user for the first command, but then we never ask for the next command!

Since we're never changing the `command` variable to anything else, it will **never** become equal to `'stop'` so the loop will never end.

Here's a simple rule to avoid infinite loops:

There must always be something *inside* the `while` loop that might change the conditon to `False`.

And of course, don't worry if you get an infinite loop! You can always hit the stop button!

8.1.5. Problem: Snooze No More!

When your alarm goes off, you can hit the *snooze* button and fall back asleep. Write a program that won't let you snooze too long!

Your program should print out `MEEP MEEP MEEP` and then keep asking the user `Are you up?` until the user enters `I am up!`

If the user says anything else, your program should print out `MEEP MEEP MEEP` and ask the user for input again.

Once the user says `I am up!` your program should finish by printing out `Took you long enough!`

For example:

```
MEEP MEEP MEEP
Are you up? Noooo
MEEP MEEP MEEP
Are you up? I'm sleepy
MEEP MEEP MEEP
Are you up? snooze...
MEEP MEEP MEEP
Are you up? snooze...
MEEP MEEP MEEP
Are you up? OK OK!
MEEP MEEP MEEP
Are you up? I am up!
Took you long enough!
```

Here is another example. The program should run until the user enters `I am up!` exactly:

```
MEEP MEEP MEEP
Are you up? Sssssh
MEEP MEEP MEEP
Are you up? i'm up
MEEP MEEP MEEP
Are you up? I am awake!
MEEP MEEP MEEP
Are you up? I am up!
Took you long enough!
```

Testing

- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing another simple interaction.
- ☐ Testing when the user gets up straight away.
- ☐ Testing when the user takes a long time to get up.
- ☐ Testing when the input starts with `I am up!` but is not exactly that.
- ☐ Testing a hidden case.

8.1.6. Problem: Duck, Duck, Goose!

There's a fun game often played in school playgrounds called Duck, Duck, Goose! (https://en.wikipedia.org/wiki/Duck,_duck,_goose) A player who is ***it*** walks around the outside of a circle of students tapping them lightly on the head and saying "Duck". When the person who is ***it*** taps someone and says "Goose" the *goose* must get up and chase the ***it*** person until getting back to the hole left by the *goose*. The last person to sit down is then the ***it*** person and the game starts again.

Write a program to help you play the game.

Your program should repeatedly ask the user `Pick?` until the user enters `Now`.

If the user says anything else, your program should print out `Duck` and ask the user for input again.

Once the user says `Now`, your program should finish by printing out `Goose! -- Now run!`

For example:

```
Pick? no
Duck
Pick? nup
Duck
Pick? not yet
Duck
Pick? I said no
Duck
Pick? Now
Goose! -- Now run!
```

Here is another example. The program should run until the user enters `Now` exactly:

```
Pick? yep
Duck
Pick? Yes
Duck
Pick? now!
Duck
Pick? Now
Goose! -- Now run!
```

Testing

- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing another simple interaction.
- ☐ Testing when the user picks straight away.
- ☐ Testing when the user takes a long time to choose.
- ☐ Testing when the input starts with something very similar to `Now`.
- ☐ Testing a hidden case.

8.2. More while loops

8.2.1. Other conditions

The *condition* of the `while` loop can be anything that is a `True` or `False` question.

Here's an example where we keep repeating the loop *while* the user is typing in uppercase:

```
line = input('Enter a line: ')
while line.isupper():
  print('Please stop shouting!')
  line = input('Enter a line: ')
print('Thank you.')
```

```
Enter a line: HELLO
Please stop shouting!
Enter a line: HOW ARE YOU?
Please stop shouting!
Enter a line: oh, sorry.
Thank you.
```

8.2.2. Looking for blank lines

A pattern that you'll need to use when writing programs is to read in multiple lines **until a blank line is entered**. Here's an example of how we can do this:

```
line = input('Enter line: ')
while line:
  print('Still running')
  line = input('Enter line: ')
print('Stopped')
```

```
Enter line: line 1
Still running
Enter line: line 2
Still running
Enter line: 
Stopped
```

We've used a shortcut for the comparison here. Rather than writing `while line != ''` we can just use `while line`, which means the same thing. The `line` variable will be `True` if the string is not empty, and `False` otherwise.

8.2.3. Doing extra things each loop

Now that we can read multiple lines of input, we can do extra things each loop!

Remember, everything that should be repeated must be *inside* the `while` loop. You must indent all the lines of code which are *inside* by adding spaces at the beginning of the line.

```python
line = input('Enter a line: ')
while line:
  print('Original: ' + line)
  print('Uppercase version: ' + line.upper())
  print('Lowercase version: ' + line.lower())
  line = input('Enter a line: ')
print('Finished')
```

Like the previous example, this program will finish only when a blank line is entered by the user.

8.2.4. Problem: That's Un-American (spelling)!

You've been reading a lot of articles online and are sick of American spelling. You decide to write a program to *un-Americanise* the text.

There are [lots of differences between American and British spelling (https://en.wikipedia.org/wiki/American_and_British_English_spelling_differences)](https://en.wikipedia.org/wiki/American_and_British_English_spelling_differences), but for this program we'll focus on two major differences: *-or*/*-our* (like color/colour) and *-ize*/*-ise*. Write a program to change all instances of `or` to `our` and `ize` to `ise`.

You only need to translate lowercase letters for this problem.

Your program should read multiple lines of input from the user, translate each line, and print out the translated line. Your program should keep translating until the user enters a blank line. For example:

```
Line: Don't rely on rumor to recognize humor.
Don't rely on rumour to recognise humour.
Line: You can analyze the patterns.
You can analyze the patterns.
Line: That will help you visualize situations.
That will help you visualise situations.
Line: Vocalize your favorites!
Vocalise your favourites!
Line: 
```

Here is another example, where the changes don't always work so well:

```
Line: A citizen was honored in the neighborhood recently.
A citisen was honoured in the neighbourhood recently.
Line: The size of the prize was hard to realize.
The sise of the prise was hard to realise.
Line: There was lots to organize and names to memorize
There was lots to ourganise and names to memourise.
Line: 
```

Hint: Use the `replace` method

If you don't remember how the `replace` method works, look for it in earlier notes.

Testing

- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing a simple case where `or` is replaced with `our`.
- ☐ Testing a simple case where `ize` is replaced with `ise`.
- ☐ Testing a case with lots of changes.
- ☐ Testing a single line of input.
- ☐ Testing with no input.
- ☐ Testing a hidden case.

8.2.5. Problem: I know you are, you said you are...

Your little brother is staying at a friend's place for a few days and you're actually missing his annoying banter (https://en.wiktionary.org/wiki/I_know_you_are_but_what_am_I). Write a program to add it back into your life!

Your program should read in lines of text from the user until the user enters an empty line. If a line starts with the words `You are` (exactly), you should print out the traditional retort: `I know you are, you said you are, but what am I?`

Here's an example:

```
Line: You are probably having fun.
I know you are, you said you are, but what am I?
Line: That seems unnecessary.
Line: You are silly.
I know you are, you said you are, but what am I?
Line: You are getting on my nerves!
I know you are, you said you are, but what am I?
Line: Stop it!
Line: That's better.
Line: 
```

Here is another short example:

```
Line: Hello!
Line: You are looking well.
I know you are, you said you are, but what am I?
Line: I should have seen that coming.
Line: 
```

Testing

- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing a short interaction.
- ☐ Testing a case with lots of backtalk.
- ☐ Testing a case with lots of back and forth.
- ☐ Testing a single line of input.
- ☐ Testing another single line of input.
- ☐ Testing with no input.
- ☐ Testing a hidden case.

8.3. Counters

8.3.1. Counters

Sometimes, you want to keep track of how many times a `while` loop repeats.

We can use a *counter*, which is a variable that starts at `0` (or `1` in this case, as we are guessing once before we enter the loop) and counts up inside the loop.

Once we have this variable we can increase it by one each time through the loop by doing this:

```
counter = counter + 1
```

This takes the number that was in `counter` before, adds 1 to it, then saves it back into the variable `counter` (and overwriting the old number). We can use this to count how many times a `while` loop repeats:

```
counter = 1
guess = input('Guess my favourite colour: ')
while guess.lower() != 'yellow':
  counter = counter + 1  # Adds 1 to the counter.
  guess = input('Try again: ')
print('You got it in', counter, 'tries.')
```

```
Guess my favourite colour: red
Try again: blue
Try again: orange
Try again: yellow
You got it in 4 tries.
```

8.3.2. Building a string

Just like we were adding 1 to the counter each time through the loop, we can also add to a string each time the loop repeats.

```
message = 'e'
while message != 'eeeee':
  print(message)
  message = message + 'e'
```

```
e
ee
eee
eeee
```

This loop keeps running until the `message` string is equal to `'eeeee'` (5 `e`'s). Notice that as soon as it is equal, the loop stops, so it never ends up printing out `'eeeee'`.

If we wanted the program to print out `'eeeee'` we could add a final `print` statement like this:

```
message = 'e'
while message != 'eeeee':
  print(message)
  message = message + 'e'
print(message)
```

```
e
ee
eee
eeee
eeeee
```

The final `print` statement runs after the loop has terminated. Experiment to see what happens.

8.3.3. Problem: Michael's Medals

Which athlete has won the most Olympic medals ever? How many guesses do you need to get the answer? `Michael Phelps` (https://en.wikipedia.org/wiki/Michael_Phelps), a swimmer from the USA. He has won 28 medals, including 23 gold!

Let's write a program to see how many attempts the user needs to get the right answer! Here is an example:

```
Who has the most medals? Ian Thorpe
Nope! Guess again: Larisa Latynina
Nope! Guess again: Dawn Fraser
Nope! Guess again: Michael Phelps
Correct!
You made 3 incorrect attempt(s).
```

Your program should keep asking forever until the user enters the exact answer. However, they might get it right the first time:

```
Who has the most medals? Michael Phelps
Correct!
You made 0 incorrect attempt(s).
```

Testing

- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing an example with only one incorrect attempt.
- ☐ Testing an example which took 5 incorrect attempts.
- ☐ Testing guesses which include the correct answer.
- ☐ Testing with guesses including blank lines.
- ☐ Testing a very long list of attempts.
- ☐ Testing a hidden case.

8.3.4. Problem: What's the time Mr Wolf?

There is a schoolyard game called [What's the time, Mr Wolf? (https://en.wikipedia.org/wiki/What%27s_the_time,_Mr_Wolf%3F)](https://en.wikipedia.org/wiki/What%27s_the_time,_Mr_Wolf%3F) One person is *Mr Wolf* and stands with their eyes closed against a wall. All the other players line up at the safe line. When the game starts the players call out *"What's the time, Mr Wolf?* Starting from 1 o'clock the wolf calls out times going up by one each time. The players must walk towards the wolf that number of steps. Finally the wolf shouts *"dinner time!"* and everyone must run back to the safe line while *Mr Wolf* tries to catch them.

Let's write a program to help play the game. Here's an example:

```
What's the time Mr Wolf? not yet
1 o'clock
What's the time Mr Wolf? not close enough
2 o'clock
What's the time Mr Wolf? getting closer
3 o'clock
What's the time Mr Wolf? dinner time
Run!!!
```

Your program should keep asking forever until the user enters the exact phrase `'dinner time'`. However, they might enter it the first time:

```
What's the time Mr Wolf? dinner time
Run!!!
```

Hint

To print the word `"What's"` or `"o'clock"`, you will need to use double quotes:

```
msg = input("What's the time Mr Wolf? ")
```

Testing

- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing an example with only one step to get to dinner time.
- ☐ Testing an example which took 5 steps to get to dinner time.
- ☐ Testing guesses which include dinner time.
- ☐ Testing with lines that include blank lines.
- ☐ Testing a very long list of steps before dinner time.
- ☐ Testing a hidden case.

8.4. Making decisions inside a loop

8.4.1. Making decisions inside a loop

Using `if` statements, we can write this short program:

```
line = input('Enter a line: ')
if 'cat' in line:
  print('I see a cat!')
else:
  print('No cat.')
```

But instead of using this `if` statement just once, we want to repeatedly ask for another line and check it for cats too.

```
line = input('Enter a line: ')
while line:
  if 'cat' in line:
    print('I see a cat!')
  else:
    print('No cat.')
  line = input('Enter a line: ')
```

```
Enter a line: the cat is asleep
I see a cat!
Enter a line: cats are everywhere!
I see a cat!
Enter a line: something about a dog
No cat.
Enter a line:
```

In this example we've put our `if` and `else` *inside* a `while` loop so that we can use it many times.

8.4.2. Indenting is important

Look very closely at the spaces at the beginning of the lines in this example:

The `if` and `else` blocks are indented because they are *inside* the `while` block.

```
line = input('Enter a line: ')
while line:
  if 'cat' in line:
    print('I see a cat!')
  else:
    print('No cat.')
  line = input('Enter a line: ')
```

For the `print` lines which are *inside* both the `while` loop and the `if` statement, we have to indent them again one more time.

Notice how the second `input` line is only indented once? That's because it's inside the `while` block, but is not inside the `else` block. This means it doesn't matter if there is or isn't a cat in the line, the program will ask for the next line anyway.

8.4.3. Problem: TL;DR

TL;DR (https://en.wikipedia.org/wiki/TL;DR) is short for *too long; didn't read*. A reader can use tl;dr to say a story is too long. An author can use tl;dr before a brief summary so readers can decide to read the full story or not.

Write a (rude!) program to complain when a line of input is too long.

Your program should read in multiple lines of input until a blank line is entered. If the line is longer than 30 characters, your program should output `TL;DR`. For lines that are 30 characters or shorter, your program should output `I read it.`

Here is an example of how your program should work:

```
Sentence: This is a short sentence.
I read it.
Sentence: This is a very long sentence and it might not get read.
TL;DR
Sentence:
```

Here is a longer example:

```
Sentence: Oh my, what a beautiful day it is today!
TL;DR
Sentence: The warm weather is delightful.
TL;DR
Sentence: It is sunny today.
I read it.
Sentence: The weather today's fantastic!
I read it.
Sentence:
```

Hint: Use `len` to find the length of a string.

You can test the length of a string like this:

```
s = input('Type something: ')
if len(s) > 10:
  print('That was more than 10 characters long.')
```

Testing

- ☐ Testing the first line of the first example in the question.
- ☐ Testing the whole first example from the question.
- ☐ Testing the first line of the second example from the question.
- ☐ Testing the whole second example from the question.
- ☐ Testing a different set of sentences.
- ☐ Testing lots of long messages.
- ☐ Testing a lot of short messages.
- ☐ Testing when the user enters a blank line straight away.
- ☐ Testing a hidden case.

8.4.4. Problem: Will the name fit?

You have a new badge printer and it only allows you to print 20 characters including spaces on each badge. That's not actually long enough for many names.

Write a program to test whether a name or nickname is too long to be printed onto a badge.

Your program should read in multiple lines of input until a blank line is entered. If the line is longer than 16 characters, your program should output `Shorten that name.` For lines that are 16 characters or shorter, your program should output `That will fit nicely.`

Here is an example of how your program should work:

```
Name: Sherlock Holmes
That will fit nicely.
Name: Dr John Watson
That will fit nicely.
Name: Prof James Moriarty
Shorten that name.
Name:
```

Here is a longer example:

```
Name: Luke Skywalker
That will fit nicely.
Name: Leia Organa
That will fit nicely.
Name: Han Solo
That will fit nicely.
Name: Lando Calrissian
That will fit nicely.
Name: Emperor Palpatine
Shorten that name.
Name:
```

Hint: Use `len` to find the length of a string.

You can test the length of a string like this:

```
s = input('Type something: ')
if len(s) > 10:
  print('That was more than 10 characters long.')
```

Testing

- ☐ Testing the first line of the first example in the question.
- ☐ Testing the whole first example from the question.
- ☐ Testing the last line of the second example from the question.
- ☐ Testing the whole second example from the question.
- ☐ Testing a different set of names.
- ☐ Testing lots of long names.
- ☐ Testing a lot of short names.
- ☐ Testing when the user enters a blank line straight away.

- [ ] Testing a hidden case.

PROJECT 3

9.1. More word games

9.1.1. Making simple games

Now that we've learnt how to repeat things, we can make our games much more detailed and fun to play!

Let's revisit the Taboo and Word Chain questions, and turn them into fully-fledged games.

9.1.2. Problem: Do you want to play questions?

Questions (https://en.wikipedia.org/wiki/Questions_(game)) is a game played by maintaining a dialogue of only questions for as long as possible.

Write a program that reads in each line of dialogue and checks that it is a question, printing `Statement!` and ending the game if a line is not a question. In this game, we will assume that everything ending in a question mark (`'?'`) is a question.

Your program should work like this:

```
line: Do you want to play questions?
line: How long does it take?
line: Do you need to go somewhere?
line: No.
Statement!
```

Here is another example:

```
line: What time is it?
line: Don't you have a watch?
line: Is there a clock somewhere?
line: Is there one in that room?
line: I don't think so.
Statement!
```

Testing

- ☐ Testing the first example.
- ☐ Testing the second example.
- ☐ Testing an example with two valid questions.
- ☐ Testing a case with '?' in a statement.
- ☐ Testing a long game.
- ☐ Testing for termination after reading a statement.
- ☐ Testing a game where no questions are asked.
- ☐ Testing a hidden case.

9.1.3. Problem: Taboo Part Two!

[Taboo (https://en.wikipedia.org/wiki/Taboo_(game))](https://en.wikipedia.org/wiki/Taboo_(game)) is a word game where one person describes a word on a card so that their partner can guess it. However, there are certain words that the person can't say when trying to describe the thing! A player might have to describe *'cereal'* without using the word *'breakfast'*.

Write a program that reads in the Taboo word, and continues to read in the description line until the line includes the taboo word. Here's an example for trying to guess the word *'cereal'*:

```
Taboo word: breakfast
Line: A type of grain
Safe!
Line: you might pour milk on it...
Safe!
Line: Museli is a type of breakfast blank.
Taboo!
```

The program should keep running until the line includes the taboo word (ignoring case):

```
Taboo word: bike
Line: there's a song about them on a bus.
Safe!
Line: where they go round and round?
Safe!
Line: Umm. - Oh! There are 4 of them on a car
Safe!
Line: and one in a unicycle
Safe!
Line: and two on a motorbike
Taboo!
```

Here's a short example when trying to guess the word *'wheel'*:

```
Taboo word: bike
Line: The things on a BIKE that go flat...
Taboo!
```

Sometimes unrelated words include the sub-word. That still counts as a Taboo! E.g. guessing *'cat'*:

```
Taboo word: meow
Line: An animal with whiskers
Safe!
Line: As a homeowner they might make your house smell
Taboo!
```

Testing

- ☐ Testing the words in the first example in the question.
- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing the fourth example in the question.

- ☐ Testing a game trying to guess *'water'*.
- ☐ Testing a similar *'water'* game but with different case input.
- ☐ Testing another example with different case input.
- ☐ Testing a hidden test case (to make sure your program works in general).
- ☐ Testing another hidden case.

9.1.4. Problem: Word chain!

Word Chain (https://en.wikipedia.org/wiki/Word_chain) is word game where players take turns saying words that start with the last letter of the previous word. You might have played this game on long car trips.

Write a program to help you play word chain. Your program should read in words until a blank line is entered. It should print out `Invalid word` if a word is not a valid play. Your program should work for upper case and lower case words.

Here is an example:

```
Word: carrot
Word: tomato
Word: orange
Word: mandarin
Invalid word
Word: eggplant
Word: 
```

Notice that the word **`mandarin`** is rejected because it doesn't start with the letter **`e`** from the previous word: **`orange`**. The next word still needs to start with the letter **`e`** (from **`orange`**), rather than **`n`** (from the end of the invalid word, **`mandarin`**).

Here is another example:

```
Word: tomato
Word: okra
Word: asparagus
Word: seaweed
Word: cake
Invalid word
Word: dried apricots
Word: cake
Invalid word
Word: 
```

Here's one last example. Don't forget it should work regardless of case!

```
Word: Australia
Word: Antartic
Word: Canada
Word: England
Invalid word
Word: Denmark
Invalid word
Word: 
```

You will always read in at least two words.

> **Hint: previous and current lines**
>
> 1. Read two lines of input before you start looping.
> 2. Notice that the previous line becomes the current line in the next iteration.

Testing

- ☐ Testing the first example from the question.
- ☐ Testing the second example from the question.
- ☐ Testing the third example from the question.
- ☐ Testing a perfect word chain.
- ☐ Testing a word chain with multiple mistakes.
- ☐ Testing a case with lots of tricky upper and lower case words.
- ☐ Testing hidden case number 1.
- ☐ Testing hidden case number 2.
- ☐ Testing hidden case number 3.

PROJECT 4: PUTTING IT ALL TOGETHER

10.1. More word games

10.1.1. Making simple games

We've now learnt how to make lots of quite complex word games! Nice work!

For our last project, we're going to put everything we've learnt together to make a chatterbot, which is a robot that can reply to your messages in a seemingly intelligent way.

We'll see if we can make our chatbot smart enough that it might just be able to trick a human into thinking it was alive.

The question of whether or not you be able to tell if you were texting your friend or a computer is trickier than you might think!

10.1.2. Problem: Introducing Captain Featherbot

Ya-har. Captain Featherbot is pleased to make your acquaintance!

Let's start off with our chatterbot. Captain Featherbot should introduce themselves, ask for the user's name, and then ask them what's on their mind. The Captain should keep letting them talk until the user enters `go away` (or `GO AWAY`).

Here's an example interaction:

```
Arrr, I am Captain Featherbot.
What be your name?
> Sarah
Ahoy Sarah! What be on your mind?
> This is a strange way to talk to a pirate
Arrr. Go on...
> But it's fun!
Arrr. Go on...
> For a little bit.
Arrr. Go on...
> go away
Shiver me timbers!
Farewell Sarah, yer landlubber.
I will be off for more swashbuckling adventures!
```

Input on another line!

Careful! In these questions, you'll need to use `input` like this, printing out a question then using a prompt string afterwards:

```
print('What be your name?')
name = input('> ')
```

We've given you a start to get you going.

Here's another example:

```
Arrr, I am Captain Featherbot.
What be your name?
> Felix
Ahoy Felix! What be on your mind?
> I want to go sailing.
Arrr. Go on...
> The ocean looks so pretty.
Arrr. Go on...
> But I don't have a boat.
Arrr. Go on...
> I need a boat to go sailing.
Arrr. Go on...
> Do you have a boat?
Arrr. Go on...
> Will you answer my question?
Arrr. Go on...
```

Notice that the user has to type *exactly* `go away` (or `GO AWAY`) without any other characters on the line for the Captain to stop talking.

You'll need

program.py

```
print('Arrr, I am Captain Featherbot.')
print('What be your name?')
name = input('> ')



print('Shiver me timbers!')
print(f'Farewell {name}, yer landlubber.')
print('I will be off for more swashbuckling adventures!')
```

Testing

- ☐ Testing the words in the first example in the question.
- ☐ Testing the capitalisation, punctuation and spaces in the first example in the question.
- ☐ Testing the words in the second example in the question.
- ☐ Testing the capitalisation, punctuation and spaces in the second example in the question.
- ☐ Testing a short example.
- ☐ Testing a longer example.
- ☐ Testing an example with tricky case.
- ☐ Testing a hidden case.

10.1.3. Problem: Captain Featherbot and the Sea

Let's make Captain Featherbot a bit more interesting to talk to.

Take your answer from the last question, and then we'll add to it in this question!

Captain Featherbot loves talking about their boat and the sea. If you mention the word `'boat'` or `'sea'` while you're talking to them, they will derail the conversation!

Add to your program, such that if you use the word `'boat'` in the line, the Captain says: `Oh, I do love my boat, Floaty McFloatface.` Or, if you use the word `'sea'` in the line, the Captain says: `Oh, the sea. Arrr to be back on the sea.`

Here's an example interaction:

```
Arrr, I am Captain Featherbot.
What be your name?
> Guybrush
Ahoy Guybrush! What be on your mind?
> I have a little row boat
Oh, I do love my boat, Floaty McFloatface.
> That's a nice name for a boat.
Oh, I do love my boat, Floaty McFloatface.
> Yes, you said you like it.
Arrr. Go on...
> But I've never been to the seaside.
Oh, the sea. Arrr to be back on the sea.
> I'd like to go!
Arrr. Go on...
> You're no help!
Arrr. Go on...
```

Here's another example. Make sure your program works with upper case and lower case!

```
Arrr, I am Captain Featherbot.
What be your name? Ahmed
Ahoy Ahmed! What be on your mind?
> BOATS! I like boats.
Oh, I do love my boat, Floaty McFloatface.
> That's a funny name for a boat.
Oh, I do love my boat, Floaty McFloatface.
> Are boats all you ever talk about?
Oh, I do love my boat, Floaty McFloatface.
> STOP TALKING ABOUT YOUR BOAT
Oh, I do love my boat, Floaty McFloatface.
> go away
Shiver me timbers!
Farewell Ahmed, yer landlubber.
I will be off for more swashbuckling adventures!
```

Boats or Seas? Which first?

You won't be given a line that has both `'boat'` and `'sea'` in it, so don't worry about which way the tests are ordered.

You'll need

program.py

```
# Use your answer for the previous question as a starting place!
```

Testing

- ☐ Testing the words in the first example in the question.
- ☐ Testing the capitalisation, punctuation and spaces in the first example in the question.
- ☐ Testing the words in the second example in the question.
- ☐ Testing the capitalisation, punctuation and spaces in the second example in the question.
- ☐ Testing a short example.
- ☐ Testing a longer example.
- ☐ Testing an example with tricky case.
- ☐ Testing a hidden case.

10.1.4. Problem: Captain Featherbot, are you listening?

Now Captain Featherbot is starting to have a bit of personality!

Take your answer from the last question, and then we'll add to it in this question!

For lines that end in a question mark, the Captain says: `That be the real question <name>. I wish I knew.` Make sure to replace `<name>` with the user's name!

If the line ends in an exclamation mark, the Captain says: `Yo ho ho. That be a good one, <name>! Then what?`

If the line starts with `I feel`, the Captain says: `When I feel that way, I go sailing. What do you do?`

Here's an example interaction:

```
Arrr, I am Captain Featherbot.
What be your name?
> Bluebeard
Ahoy Bluebeard! What be on your mind?
> I'm a pirate!
Yo ho ho. That be a good one, Bluebeard! Then what?
> No, that's it.
Arrr. Go on...
> Are you following me?
That be the real question Bluebeard. I wish I knew.
> Don't follow me!
Yo ho ho. That be a good one, Bluebeard! Then what?
> I feel like I'm not being understood.
When I feel that way, I go sailing. What do you do?
> I try to explain it a different way.
Arrr. Go on...
```

Which rule first?

You won't be given a line that matches multiple criteria, so don't worry about which way the rules are ordered.

You'll need

program.py

```
# Use your answer for the previous question as a starting place!
```

Testing

- ☐ Testing the words in the first example in the question.
- ☐ Testing the capitalisation, punctuation and spaces in the first example in the question.
- ☐ Testing a short example.
- ☐ Testing a longer example.
- ☐ Checking the checks for `?` and `!` are precise.
- ☐ Testing a hidden case.

10.1.5. Problem: Captain Featherbot!

Take your answer from the last question, and then we'll add to it in this question!

There's one more thing to add to Captain Featherbot's repertoire.

If the user enters a line that starts with `I am <something>`, the Captain should say: `When I was last <something> I stole a boat and sailed the seas.` The Captain will replace the `<something>` with what the user typed in.

This should work if the user enters a line that starts with that pattern regardless of the case of the letters.

Here's an example interaction:

```
Arrr, I am Captain Featherbot.
What be your name?
> Beets McGee
Ahoy Beets McGee! What be on your mind?
> I am hungry
When I was last hungry I stole a boat and sailed the seas.
> i am not amused
When I was last not amused I stole a boat and sailed the seas.
> go away
Shiver me timbers!
Farewell Beets McGee, yer landlubber.
I will be off for more swashbuckling adventures!
```

Here's another example.

```
Arrr, I am Captain Featherbot.
What be your name?
> Captain Bitbots
Ahoy Captain Bitbots! What be on your mind?
> Bits and bytes
Arrr. Go on...
> I want a boat.
Oh, I do love my boat, Floaty McFloatface.
> Why did you name it that?
That be the real question Captain Bitbots. I wish I knew.
> You don't know?!
Yo ho ho. That be a good one, Captain Bitbots! Then what?
> I feel like you should know.
When I feel that way, I go sailing. What do you do?
> I go look at the sea.
Oh, the sea. Arrr to be back on the sea.
```

Putting it all together!

This question's tricky! There's a lot to put together. Stick with it and you'll be hacking like a pirate it no time!

You'll need

program.py

```
# Use your answer for the previous question as a starting place!
```

Testing

☐ Testing the words in the first example in the question.

- ☐ Testing the capitalisation, punctuation and spaces in the first example in the question.
- ☐ Testing the words in the second example in the question.
- ☐ Testing the capitalisation, punctuation and spaces in the second example in the question.
- ☐ Testing a short example.
- ☐ Testing a longer example.
- ☐ Testing an example with tricky case.
- ☐ Testing a hidden case.
- ☐ **Yar! Great job, me hearties! Time to celebrate!**

10.1.6. Congratulations, me hearties!

Arrrrr! I be impressed!

Excellent work on finishing the project, and the course! You've learnt how to solve problems with code, and make a cheeky chatbot as well!

We hope you enjoyed this course, and can't wait to see what swashbuckling adventures await you!

10.1.7. Problem: Chatbot Playground!

What's this?! You thought you were finished?

Why not have a go at writing your very own chatbot! You can write whatever code you like in this question. Consider it your personal chatbot playground!

> **Save or submit your code!**
> There are no points to be earned for this question, so you can submit whatever code you like. Make sure you save programs that you want to keep!

Testing

- [ ] This question is a playground question! There's no right or wrong.