Australian
Computing
Academy

DT Challenge Python

Year 7 Biology

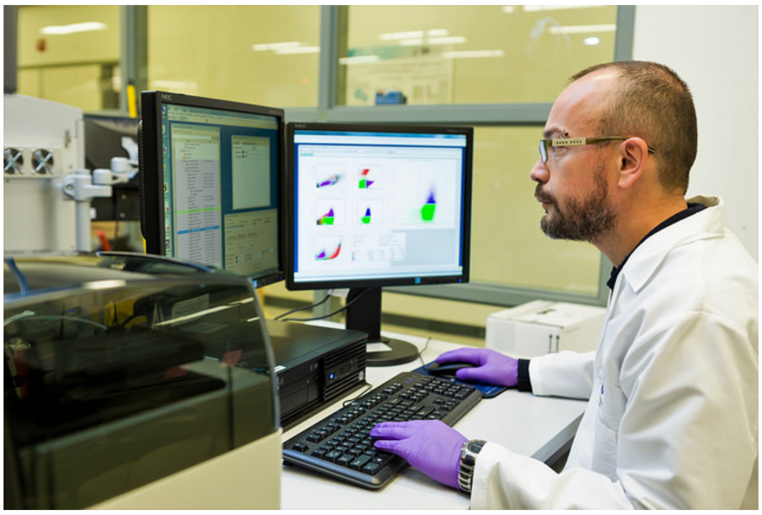
1 OUTPUT

1.1. Getting started

1.1.1. Why biology and Python?

In this course you'll learn two things:

- how biologists classify living things; and
- writing computer programs with Python


Biologists use computers to analyse data. Daniel Soñé Photography, LLC (https://www.flickr.com/photos/64860478@N05/38997905444/) CC BY 2.0

Scientists rely on computers to do lots of data processing and analysis for them. This data processing is all performed with programs written in languages like Python!

1.1.2. Hello, Biology!

Normally, the first program you write when learning a new programming language is Hello, World! (https://en.wikipedia.org/wiki/Hello_world_program), but we're going to write a slightly different one instead:

```python
print('Hello, Biology!')
```

```
Hello, Biology!
```

You can edit and run any example in Grok by clicking the ▶ button. Try changing 'Hello, Biology!' 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.

1.1.3. 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, Biology!')
```

```
Hello, Biology!
```

Purple tells you that `print` is a *function* and green tells you that `'Hello, Biology!'` 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.

1.1.4. 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 in red (it does not recognise the name `write`), 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.

1.1.5. A SyntaxError?

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.

1.1.6. Problem: Hello, Biology!

Write a program that prints out the message:

```
Hello, Biology!
```

If you're not sure how to start writing the program, go back a few slides 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!**

1.1.7. It pays to be careful

Did you forget to use capital letters or match the punctuation exactly? You might be thinking that it doesn't matter, because the program did what it was expected to do.

Every time you write a program you need to be very careful - tiny mistakes like using the wrong coordinates (https://www.reuters.com/article/us-space-launch-russia-mistake/russia-says-satellite-launch-failure-due-to-programming-error-idUSKBN1EL1G2) can lead to big, expensive problems. Always match your input and output *exactly* for all of the problems in this challenge.

You can copy and paste any example code into any editor window. Try copying the example output below into the code editor, and run the code to check the output matches.

```
Hello, Biology!
```

```python
print('the output here')
```

```
the output here
```

By copying and pasting expected input and output, you can reduce the likelihood you'll make a mistake with capitalisation, punctuation or spelling.

Use this in questions!

By doing this in questions, you can be confident your answer will meet the requirements!

1.1.8. Problem: Do you know Mr DNA?

In Jurassic Park (http://www.imdb.com/title/tt0107290/), the park scientists were able to re-create extinct animals using their DNA (https://en.wikipedia.org/wiki/DNA).

Scientists don't believe this would actually be possible (http://www.iflscience.com/technology/could-jurassic-park-ever-come-true/), but DNA does tell us how living things have changed over time.

DNA has a double-helix structure and contains your unique genetic code

In the movie there is an educational video to explain how scientists used DNA to bring dinosaurs back to life. The video introduces Mr. DNA with the following phrase:

```
Oh, Mr. DNA! Where did you come from?
```

Write a program that prints this same welcome to the screen.

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

> **You don't have to type this out!**
> Remember what we said on the last slide about making sure your question meets the requirements exactly?

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. Printing multiple times

1.2.1. Why classify?

Classification involves sorting things into a group.

There are many different ways to classify. For example, you could classify lollies according to colour or taste.

How would you classify your jellybeans?

Libraries classify books using a Dewey Decimal number (https://en.wikipedia.org/wiki/Dewey_Decimal_Classification), so that all books on a particular subject are grouped together. It's much easier to find something that way than keeping books together based on colour of cover!

In biology, an *organism* is a living thing, and the classification of organisms is called *taxonomy*.

1.2.2. A string of characters

When we write things to the screen, we need to let the computer know how to understand it. We do this by using a *string*.

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

```
print('There are 4 species of kangaroo!')
```

```
There are 4 species of kangaroo!
```

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("'Neigh!' - from the horse's mouth!")
```

```
'Neigh!' - from the horse's mouth!
```

`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.

1.2.3. Printing multiple times

Our programs would be pretty boring if they only printed one thing!

We can add multiple `print` statements to print multiple lines! Python will run the statements in order, so to print multiple lines you can use:

```
print('An organism is a living thing.')
print('Classification of organisms is called taxonomy.')
```

```
An organism is a living thing.
Classification of organisms is called taxonomy.
```

Each message will be printed on it's own line, **run the example to check**!

1.2.4. 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:

```
organisms = 'An organism is a living thing.'
print(organisms)
print(organisms)
print(organisms)
print('Classification of organisms is called taxonomy.')
```

```
An organism is a living thing.
An organism is a living thing.
An organism is a living thing.
Classification of organisms is called taxonomy.
```

The first statement creates a new variable called `organisms` and stores the string `'An organism is a living thing.'` We then `print` the value of the `organisms` variable three times.

1.2.5. F-strings

Our variables in Python store information, but sometimes we want to put that information *into* a string.

Python let's us do that using f-strings! Let's take a look at how it works:

```
science = 'Biology'
print(f'{science} is fun!')
```

```
Biology is fun!
```

The `f'{science}'` is what does the magic for us. The `f` in front of the string tells python this is an *f-string*, and the curly braces `{science}` means put the contents of the `science` variable here!

Compare that to a regular string:

```
science = 'Biology'
print('{science} is fun!')
```

```
{science} is fun!
```

F-strings let us insert variables into strings!

1.2.6. Problem: Organisms

Sometimes, it's hard to remember a fact or statement. One way to do it is to repeat it multiple times so that you can remember!

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 `remember`.

Update this program so that it works for a different remember: `'Classifying organisms is taxonomy.'` Your updated program should print the message:

```
Classifying organisms is taxonomy.
Classifying organisms is taxonomy.
Classifying organisms is taxonomy.
Classifying organisms is taxonomy.
Classifying organisms is taxonomy.
I will remember! Classifying organisms is taxonomy.
```

Only change the `remember` variable!

You just need to change the value of `remember` to be `'Classifying organisms is taxonomy.'` instead of `'An organism is a living thing.'`, run it to check it works, and then mark it.

You'll need

program.py

```python
remember = 'An organism is a living thing.'
print(remember)
print(remember)
print(remember)
print(remember)
print(remember)
print(f'I will remember! {remember}')
```

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!**

1.2.7. Problem: Non-living things

In Biology, there are three categories we can use to classify all things

- living - alive
- non-living - was never alive
- dead - no longer alive.

There are lots of important non-living things, like sunlight, water and air. These are things that enable life to exist, but are not living things themselves.

We've put a program in the editor that has a variable, `non_living`, that contains information to be printed.

Click ▶ Run to see what it does.

Mistake! The program is about a `zombie`, which is a fictional character that doesn't exist, and wouldn't be biologically correct even if it did.

Update the program to bring it back to reality to work for a different object, a `rock`.

Your updated program should print the message:

```
A rock was never alive.
It's neither living nor dead, a rock is non-living.
```

Only change the `non_living` variable!

You just need to change the value of `non_living` to be `rock` instead of `zombie`, run it to check it works, and then mark it.

You'll need

program.py

```python
non_living = 'zombie'
print(f'A {non_living} was never alive.')
print(f"It's neither living nor dead, a {non_living} is non-living.")
```

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, you rock!**

1.3. Classification

1.3.1. Problem: Classification

The classification of organisms is known as...

- ○ Organising
- ○ Categorising
- ○ Taxonomy
- ○ Labelling

Testing

☐ That's right!

1.3.2. Problem: The study of living organisms

Which of the following code snippets will correctly print the message below on the screen?

```
Biology is the study of living organisms.
```

- ○
```
branch = "Biology"
print(f"branch is the study of living organisms.")
```

- ○
```
branch = "Biology"
print(f"Physics is the study of living organisms.")
```

- ○
```
branch = "Biology"
print(f"{branch} is the study of living organisms.")
```

- ○
```
branch = "Biology"
print(f"{branch}is the study of living organisms.")
```

Testing

☐ That's right!

2

INPUT

2.1. Reading user input

2.1.1. 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:

```python
species = 'Homo sapiens!'
print(species)
species = 'Human!'
print(species)
```

The old value in `species` - `'Homo sapiens'` - is replaced with the new value - `'Human!'` - before the second `print` statement, producing:

```
Homo sapiens!
Human!
```

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.

2.1.2. Asking the user a question

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

```python
animal = input('Can you name an animal? ')
print(animal)
```

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:**

```
Can you name an animal? Duck
Duck
```

The program prints the prompt `Can you name an animal?` and waits for the user to type in an animal and press `Enter`. 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, an animal).

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

2.1.3. 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 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('I love Biology!')
```

```
I love Biology!
```

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)
```

2.1.4. Problem: Lyrebird mimic

The Lyrebird (https://en.wikipedia.org/wiki/Lyrebird) is an Australian bird that has the ability to mimic sounds it hears in its environment. It can mimic not only natural sounds but artifical ones too (https://youtu.be/VjE0Kdfos4Y?t=58s)! It's also on the 10c coin.

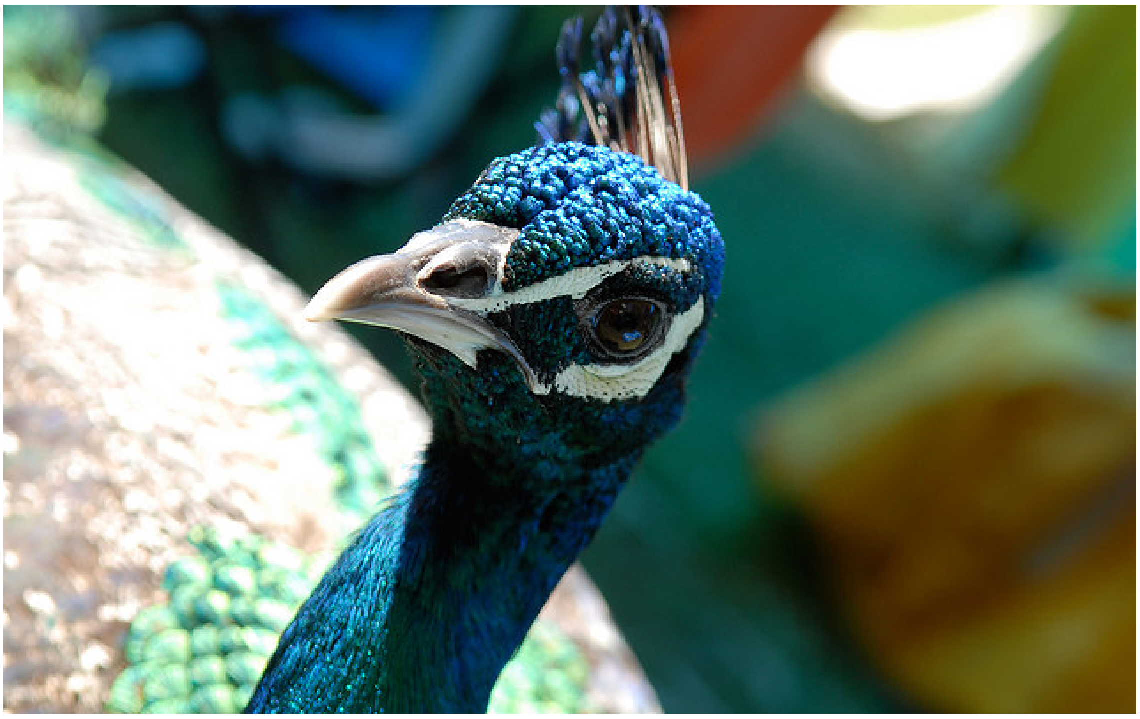


Photo by Leon Wilson CC-BY-2.0 (https://creativecommons.org/licenses/by/2.0/)

Write a program that simulates the lyrebird's mimic ability. You need to get input from the user and print back exactly what the user entered.

```
What sound do you want to make? Squawk!
Squawk!
```

Your program should work with anything the user types:

```
What sound do you want to make? Blurrgh!
Blurrgh!
```

No f-string required

You don't need f-strings here because we're just printing the variable out on its own.

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 `Squawk!`
- ☐ Testing with the word `Blurrgh!`
- ☐ Testing a hidden test case (to make sure your program works in general).

2.1.5. 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:

```python
animal = input('Favourite animal? ')
print(f'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.

2.1.6. Problem: The animal is...

Write a program to take an input species, and say it back to us:

```
Animal: human
The animal you wrote was: human
```

Your program should work with any animal name:

```
Animal: pig
The animal you wrote was: pig
```

When you run your program, it should wait for you to type in the animal 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 your output are correct.
- ☐ Testing that the punctuation in your output is correct.
- ☐ Testing that the capitalisation in your output is correct.
- ☐ Testing that the whitespace in your output is correct.
- ☐ **Yay, you've got the first example right!**
- ☐ Testing the second example in the question (when the user enters `pig`).
- ☐ Testing with the animal `giraffe`.
- ☐ Testing a two word animal (`saltwater crocodile`).
- ☐ Testing a hidden test case (to make sure your program works in general).
- ☐ Testing another hidden test case.

2.1.7. Problem: What we saw at the zoo

There are lots of different things to see at the zoo and you want to remember all the things you saw! Write a program that prints out a sentence you can copy and paste into your diary for any of the things you saw.

```
What did we see? elephants
We saw elephants at the zoo!
```

Your program should work with any friend's name:

```
What did we see? wombats
We saw wombats at the zoo!
```

When you run your program, it should wait for you to type in what you saw, using `input`, then use the same thing 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 output sentence are correct.
- ☐ Testing that the punctuation in the output sentence is correct.
- ☐ Testing that the capitalisation in the output sentence is correct.
- ☐ Testing that the whitespace in the output sentence is correct.
- ☐ **Yay, you've got the first example right!**
- ☐ Testing the second example in the question (when the user enters `wombats`).
- ☐ Testing when you saw `people`.
- ☐ Testing something with two words (`tasmanian devils`).
- ☐ Testing a hidden test case (to make sure your program works in general).
- ☐ Testing another hidden test case.

2.2. More with variables

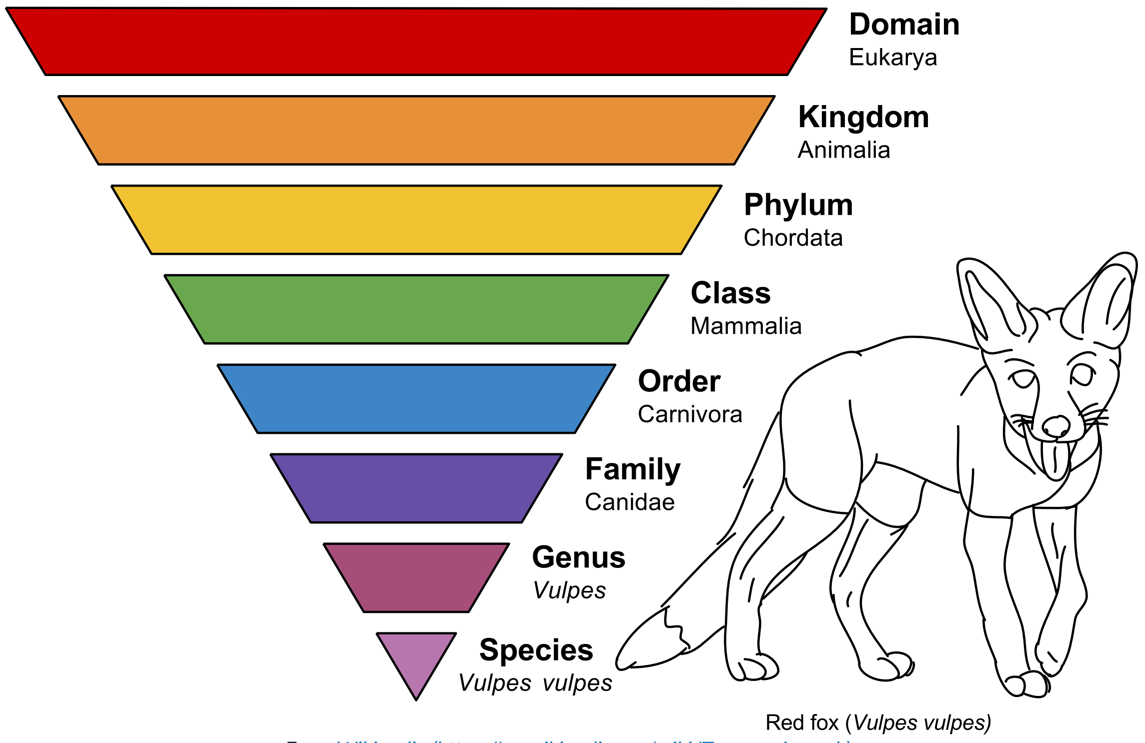
2.2.1. Taxonomy

The taxonomic rank (https://en.wikipedia.org/wiki/Taxonomic_rank) categorises each organism into 8 distinct categories.

Those are:

- Domain
- Kingdom
- Phylum
- Class
- Order
- Family
- Genus
- Species

With each step down in classification, organisms are split into more specific groups.



From Wikipedia (https://en.wikipedia.org/wiki/Taxonomic_rank).

2.2.2. Common and scientific name

Each species can have up to two names.

- *common name* - what we use in regular speech,
- *scientific name* - to classify each organism exactly.

For example, the common name *kangaroo* doesn't describe a species *exactly*. There are many different types of kangaroo, and it also doesn't describe how the species relates to similar animals, like the wallaby.

So instead, we can use the *scientific name*, which is usually made up of the *genus* and *species* in the taxonomic rank.

The Eastern Grey Kangaroo has the scientific name **Macropus giganteus**. The red-necked wallably is **Macropus rufogriseus**. See how the genus of both is *Macropus*?*

We can write a program to print these!

```python
genus = 'Macropus'
species = 'giganteus'
print(f'Scientific name: {genus} {species}')
```

```
Scientific name: Macropus giganteus
```

* *It turns out that wallabies and kangaroos are of the same genus (Macropus), but rock-wallabies are a different genus (Petrogale)! And all of them are from the Macropodidae family.*

2.2.3. Using multiple variables

When you need to store different bits of information, you can create more variables. As long as they have different names (otherwise, you're setting an existing variable).

```python
domain = 'Eukarya'
kingdom = input('What is the kingdom? ')
phylum = input('What is the phylum? ')
highest_ranks = f'{domain}, {kingdom}, {phylum}'
print('The highest taxonomic ranks are: ' + highest_ranks)
```

The variable names make it very clear what is stored in each! We can set the `domain` variable to a fixed value, since for this example we're using animals and all animals are in the `'Eukarya'` domain.

```
What is the kingdom? Animalia
What is the phylum? Chordata
The highest taxonomic ranks are: Eukarya, Animalia, Chordata
```

Notice how we can use variables that have assigned values or `input` values together? We can treat all string variables the same way when printing.

2.2.4. Problem: Scientific name generator

Scientific names of an organism often consists of the *genus* and *species* joined together, with a space in between.

Write a program that asks for each, then prints out the scientific name!

Here is an example:

```
What is the genus? Homo
What is the species? sapiens
The scientific name is Homo sapiens!
```

Here's another example:

```
What is the genus? Setonix
What is the species? brachyurus
The scientific name is Stonix brachyurus!
```

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 a red kangaroo.
- ☐ Testing an emu.
- ☐ Testing a hidden case.

2.2.5. Problem: Common genus

When two species have the same genus, they are very closely related. For example, the scientific name of the eastern grey kangaroo is the *Macropus giganteus* and the name for the Red-necked wallaby is *Macropus rufogriseus*.

This is called having a *common genus*.

Write a program that asks for a common genus, then *two* species names, then prints out the scientific name of each!

Here is an example:

```
What is the genus? Macropus
What is species 1? giganteus
What is species 2? rufogriseus
The first species is Macropus giganteus.
The second species is Macropus rufogriseus.
```

Here's another example, comparing the bonobo and the chimpanzee:

```
What is the genus? Pan
What is species 1? paniscus
What is species 2? troglodytes
The first species is Pan paniscus
The second species is Pan troglodytes
```

Testing

- ☐ Testing the words in the first prompt message.
- ☐ Testing the words in the second prompt message.
- ☐ Testing the words in the third 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 first match line.
- ☐ Testing the words in the second match line.
- ☐ Testing the punctuation, spaces and capitals in the match lines.
- ☐ **Testing the first example in the question.**
- ☐ Testing the second example in the question.
- ☐ Testing some rock-wallabies.
- ☐ Testing some Australian frogs.
- ☐ Testing a hidden case.

2.3. Taxonomy

2.3.1. Problem: Taxonomic rank

The correct ordering of rank (from highest to lowest) in the taxonomic hierarchy is...

○

1. Kingdom
2. Phylum
3. Class
4. Order
5. Family
6. Genus
7. Species

○

1. Kingdom
2. Phylum
3. Order
4. Class
5. Family
6. Genus
7. Species

○

1. Kingdom
2. Class
3. Family
4. Phylum
5. Genus
6. Order
7. Species

○

1. Kingdom
2. Class
3. Order
4. Phylum
5. Family
6. Genus
7. Species

Testing

☐ That's right!

2.3.2. Problem: Scientific names

The scientific name of a species is made up of its...

- ○ Genus and Species
- ○ Family and Species
- ○ Family and Genus
- ○ Order and Species

Testing

☐ That's right!

2.3.3. Problem: Name an animal

Which of the following programs would correctly generate both outputs shown below?

```
Name an animal: cassowary
The cassowary is a member of the Animalia kingdom.
```

```
Name an animal: emu
The emu is a member of the Animalia kingdom.
```

- ○
```
animal = input("cassowary")
print(f"The {animal} is a member of the Animalia kingdom.")
```

- ○
```
cassowary = input("Name an animal: ")
print(f"The cassowary is a member of the Animalia kingdom.")
```

- ○
```
animal = input("Name an animal: ")
print(f"The {animal} is a member of the Animalia kingdom.")
```

- ○
```
input("Name an animal: ")
print(f"The {input} is a member of the Animalia kingdom.")
```

Testing

- ☐ That's right!

2.3.4. Problem: Do you own a Felis catus?

Which of the following code snippets will produce the output shown below?

```
The scientific name of the common house cat is Felis catus.
It belongs to the Family Felidae.
```

○
```python
scientific_name = Felis catus
family = "Felidae"
print(f"The scientific name of the common house cat is {scientific_name}.")
print(f"It belongs to the Family {family}.")
```

○
```python
scientific_name = "Felis catus"
family = "Felidae"
print(f"The scientific name of the common house cat is {scientific}.")
print(f"It belongs to the Family {family}.")
```

○
```python
scientific_name = "Felis catus"
family = "Felidae"
print(f"The scientific name of the common house cat is {scientific_name}.")
print(f"It belongs to the Family {family}.")
```

○
```python
scientific_name = "Felis catus"
family = "Felidae"
print f"The scientific name of the common house cat is {scientific_name}."
print f"It belongs to the Family {family}."
```

Testing

☐ That's right!

3

DECISIONS

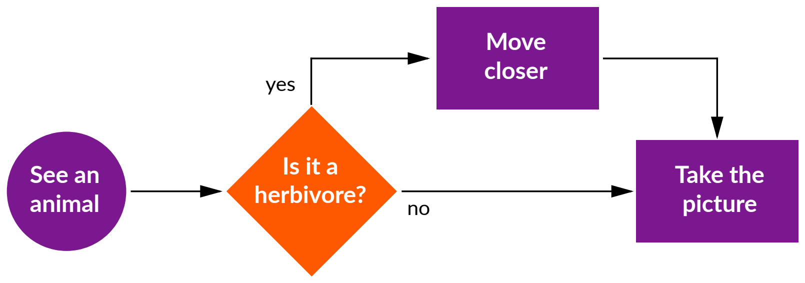
3.1. Making decisions

3.1.1. 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 we were on a safari and wanted to take a photo of an animal - we might want to know if the animal not going to eat us (a *herbivore*), before moving closer.

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 moving closer. If the answer is no `no`, we skip it.

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

3.1.2. Making decisions

Let's write our flowchart as a Python program:

```
herbivore = input('Is it a herbivore (yes/no)? ')
if herbivore == 'yes':
  print('Move a bit closer.')
print('Take the photo.')
```

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 `herbivore` *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.

3.1.3. 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:

```
animal = input('What is your favourite animal? ')
if animal == 'Tardigrade':
  print('A Tardigrade!')
  print('I wish I could survive in outer space...')
print('Cool!')
```

If the name matches `'Tardigrade'` 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:

```
if animal == 'Quokka':
  print('I love the Quokka!')
    print('You can find them on Rottnest Island!')
```

```
  File "program.py", line 3
    print('You can find them on Rottnest Island!')
    ^
IndentationError: unexpected indent
```

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

3.1.4. 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:

```
species = 'Human'
if species == 'Human':
  print("Me too! Oh wait, I'm a robot.")
```

```
Me too! Oh wait, I'm a robot.
```

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 `"Human"` 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 `'Human'` 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:

```python
species = 'Human'
if species = 'Human':
  print("Me too! Oh wait, I'm a robot.")
```

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

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

3.1.5. Problem: I love marsupials!

You're in science class and the teacher is teaching you different features of animals.

Marsupials (https://en.wikipedia.org/wiki/Marsupial) are animals that have pouches, like a kangaroo!

A kangaroo carries its young, or *joey*, in its pouch

Write a program to check the feature the teacher says. Since all animal features are pretty cool, no matter what feature it is your program still should print out `That's a cool feature.`

```
What is the feature? feathers
That's a cool feature.
```

If it's a `pouch`, the program should also print `I love marsupials!` For example:

```
What is the feature? pouch
That's a cool feature.
I love marsupials!
```

Here is another example, with a different feature:

```
What is the feature? gills
That's a cool feature.
```

Remember that you should always print out `That's a cool feature.`, whether that feature is a `pouch` or not.

Double quotes

Remember if you have a single quote *inside* the string, you will need to use *double-quotes* to display it.

For example: `"She's cool."`

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 feature (`claws`).
- ☐ Testing a different feature (`trunk`).
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

3.1.6. Problem: The "bin chicken"

You're wandering around the streets of Sydney and see a mysterious animal nearby. Lots of animals native to Australia are now forced to live in cities because humans have altered their natural environments. This can make survival difficult for these species.

Whatever the `input` from the user, your program should print `I hope the {animal} can survive out there...`, **substituting the animal's name into the output**. For example:

```
What animal is it? bat
I hope the bat can survive out there...
```

There's a good chance in large cities you'll see an ibis (https://en.wikipedia.org/wiki/Australian_white_ibis), which local residents refer to informally as the Bin Chicken (http://www.nationalgeographic.com.au/australia/magpie-beats-bin-chicken-for-aus-bird-of-the-year.aspx).

The Australian white ibis

If the animal is an `ibis`, exclaim that you've seen `A Bin Chicken!`.

```
What animal is it? ibis
A Bin Chicken!
I hope the ibis can survive out there...
```

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

```
What animal is it? quokka
I hope the quokka can survive out there...
```

The sad story of the Ibis

The *Australian White Ibis* has been forced to live off rubbish in urban environments (https://www.gizmodo.com.au/2017/11/in-defence-of-the-bin-chicken/) due to human development gradually forcing it out of its natural habitat. Eating rubbish isn't good for them, and restoring their wetlands would go a long way to improving their long-term survival.

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 animal (`kangaroo`).
- ☐ Testing a two-word animal (`alley cat`).
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

3.1.7. 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:

```python
animal = 'snake'
if animal == 'snake':
  print('Ssssssss')
```

```
Ssssssss
```

If we change the value stored in `animal` the expression will evaluate to `False` and the block of code will be skipped:

```python
animal = 'doggo'
if animal == 'snake':
  print('Ssssssss')
```

(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:

```python
animal = 'snake'
print(animal == 'snake')
```

```
True
```

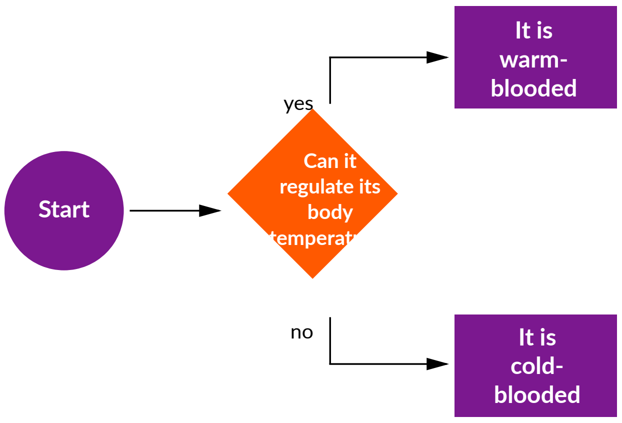
Try changing the `'snake'` on the first line above to something else and see what happens.

3.2. Decisions with two options

3.2.1. Dichotomous Keys

When sorting biological organisms (i.e. developing our taxonomy), we often make our decisions about how to classify the organism based on whether a particular feature is present or not. Because these decisions have two possible options, we refer to these decisions as **dichotomous keys**.

We can represent a dichotomous key in programming by making a decision when a condition is `True`, and doing *something else* when a condition is `False`.



A characteristic of all warm-blooded animals is their ability to regulate their own body temperature. This allows them to keep their body temperature the same even when the weather changes.

Cold-blooded animals become hotter and colder when the temperature outside changes. At night their bodies get cooler, and during the day when they are in the sun they warm up.

We could write a program that asks if the animal can regulate its body temperature. If the user says `'yes'` it can, then it tells the user it is warm-blooded. Otherwise it says it is cold-blooded.

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

3.2.2. Warm- or cold-blooded?

In Python the `else` keyword specifies the steps to follow if the condition is `False` (in this case if the animal can **not** regulate its temperature).

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

```
regulate = input('Can it regulate its body temperature? ')
if regulate == 'yes':
  print('It is warm-blooded.')
else:
  print('It is cold-blooded.')
```

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.

3.2.3. Problem: Plant or animal?

Let's create our first *dichotomous key* to classify if an organism is a plant or an animal.

Scientifically, the difference between a plant and animal is that plants have a cell wall, while animals do not.

Write to program to check if the organism is a plant or an animal, if the user types in `yes`, then it should say `Then it's a plant!`.

Here's an example:

```
Does it have a cell wall? yes
Then it's a plant!
```

If they say *anything* else, you should say, `It's probably an animal.`

```
Does it have a cell wall? no
It's probably an animal.
```

If the answer is not *exactly* `yes`, you should still print out the other message. For example, your programs does not understand slang:

```
Does it have a cell wall? yesssss
It's probably an animal.
```

Hint!

Remeber when we print a `'` in our string, we need to use double-quotes `"` around our message.

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.
- ☐ Testing a typo.
- ☐ Testing a hidden case.

3.2.4. Problem: Smells fishy

We can also classify animals based on their *features* or *characteristics*.

Let's make another *dichotomous key* to tell if an animal is a fish, or not.

When choosing a characteristic, it helps to try and be precise, for example all fish are *cold-blooded*, but that is also a characteristic of *reptiles*.

So instead, we can use *gills*. All fish have gills so they can breathe underwater.

Write a program to ask if the creature has gills, and if the answer is `yes`, then it should respond `It is a fish!`.

Here's an example:

```
Does the creature have gills? yes
It is a fish!
```

If they say *anything* else, you should say, `The creature is not a fish.`

```
Does the creature have gills? no
The creature is not a fish.
```

If the answer is not *exactly* `yes`, you should still print out the other message. For example, your programs does not understand slang:

```
Does the creature have gills? yeah
The creature is not a fish.
```

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.
- ☐ Testing a typo.
- ☐ Testing a hidden case.

3.3. Dichotomous keys

3.3.1. Problem: Dichotomous Keys

What is a *dichotomous key*?

◯ An `if`-`else` statement in Python.

◯ A taxonomic rank.

◯ A tool to identify organisms by dividing groups into two categories.

◯ An `if`-`elif`-`else` block of code.

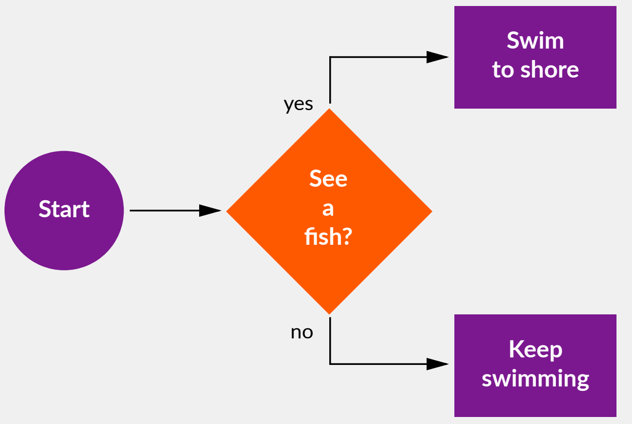
Testing

☐ That's right!

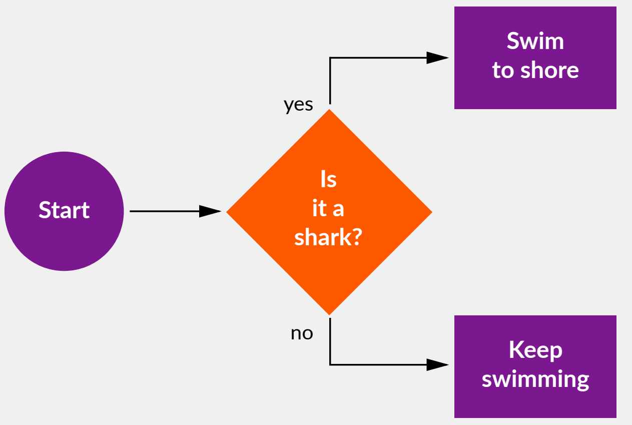
3.3.2. Problem: Is it a shark?

Choose the flowchart below that represents the following code snippet.

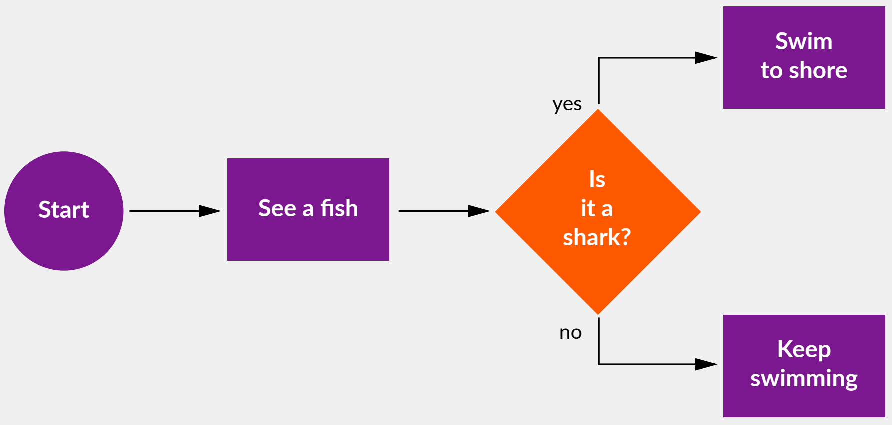
```
print('Oh look! A fish!')
shark = input('Is it a shark (yes/no)? ')
if shark == 'yes':
  print('Swim to shore!')
else:
  print('Keep swimming')
```

○ 

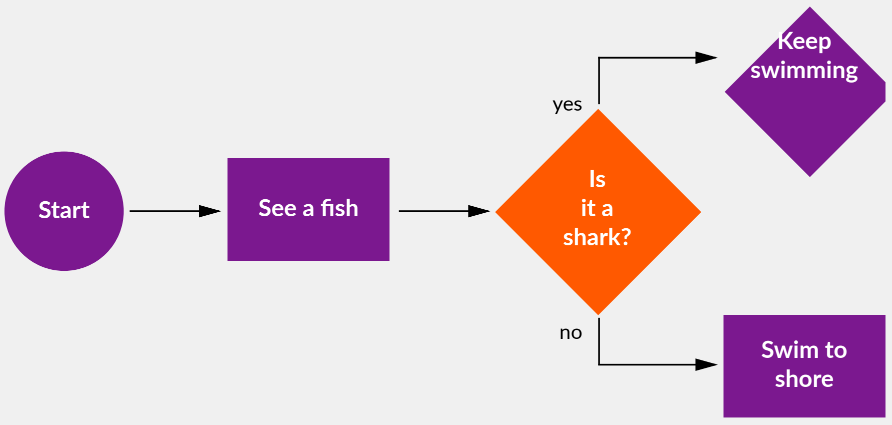
Start
See a fish?
yes
Swim to shore
no
Keep swimming

○ 

Start
Is it a shark?
yes
Swim to shore
no
Keep swimming

○ 

Start
See a fish
Is it a shark?
yes
Swim to shore
no
Keep swimming

○ 

Start
See a fish
Is it a shark?
yes
Keep swimming
no
Swim to shore

Testing

☐ That's right!

3.3.3. Problem: Australia's favourite bird

Which of the following code snippets will produce the output?

```
Name a bird? magpie
That's Australia's favourite bird!
```

```
Name a bird? swan
I like that bird.
```

```
Name a bird? tiger
I like that bird.
```

* according to [The Guardian (https://www.theguardian.com/environment/live/2017/dec/11/bird-of-the-year-150000-votes-counted-as-ibis-fans-anxiously-await-results)](https://www.theguardian.com/environment/live/2017/dec/11/bird-of-the-year-150000-votes-counted-as-ibis-fans-anxiously-await-results) poll, anyway.

○
```python
bird = input('Name a bird? ')
if bird == 'magpie':
  print('That's Australia's favourite bird!')
print('I like that bird.')
```

○
```python
magpie = input('Name a bird? ')
if bird == 'magpie':
  print("That's Australia's favourite bird!")
else:
  print("I like that bird.")
```

○
```python
bird = input('Name a bird? ')
if bird == 'magpie':
  print("That's Australia's favourite bird!")
else:
  print("I like that bird.")
```

○
```python
bird = input('Name a bird? ')
if bird == 'magpie':
  print(f"A {bird}")
else:
  print("I like that bird.")
```

Testing

☐ That's right!

4 COMPLEX DECISIONS

4.1. Decisions with more options

4.1.1. More difficult questions

Sometimes we may want to test things that can have more than two possible answers. Let's start with a simple situation where we're checking to see what kind of diet an animal has.

```python
eats = input('What type of animal are you? ')
if eats == 'carnivore':
  print('You only eat meat.')
else:
  print('You only eat plants.')
```

We've seen questions like this before where the user types in a word and we check what that word is. Here, if the user doesn't type in `'carnivore'`, the program assumes that the animal only eats plants (i.e. is a `'herbivore'`).

This isn't right! It's possible that the animal eats both meat and plants, which would make it an `'omnivore'`. We need a way for our program to deal with that!

4.1.2. More than two options

If we want to test if an animal is a *carnivore*, *herbivore* or *omnivore*, we need to add something extra to our `if` statement. Computers can only check one thing at a time and those statements always need to evaluate to `True` or `False`.

This means we need to check one condition first and, if that one is `False`, have the computer then check a different one. We use an `elif` to do this.

`elif` is an abbreviation of `else` and `if` together, and allows us to write code that will check something else if our first condition is `False`. The code below demonstrates how to use an `elif`:

```python
eats = input('What type of animal are you? ')
if eats == 'carnivore':
  print('You only eat meat.')
elif eats == 'herbivore':
  print('You only eat plants.')
else:
  print('You eat both meat and plants.')
```

The program checks the value of `eats` and, if it is `'carnivore'`, prints out the meat message and skips over the rest. If it isn't `'carnivore'`, it checks if it is `'herbivore'`, and prints out the plants message if that is `True`. If that's still not correct, it prints the message in the `else` clause (i.e. that it is an omnivore).

4.1.3. Even more options

Our previous example assumed that if you weren't a `'carnivore'` or `'herbivore'` you had to be an `'omnivore'`, but it's possible the user might type something else in that isn't correct. To fix this, we want to also check if `'omnivore'` is typed in by the user, and we can do this with another `elif`:

```python
eats = input('What type of animal are you? ')
if eats == 'carnivore':
  print('You only eat meat.')
elif eats == 'herbivore':
  print('You only eat plants.')
elif eats == 'omnivore':
  print('You eat both meat and plants.')
else:
  print('Are you an animal?')
```

You can add as many `elif` clauses as you like, to deal with all the different cases you might have to catch.

Remember that an `else` can always be used to catch anything that isn't matched by **any** of the `if` or `elif` conditions - it's a good way to make sure that you only execute the right code for correct values.

4.1.4. Problem: Feature to class

The **class** of an animal describes what general type of animal it is, such as *mammal*, *reptile* or *fish*.

Write a program to classify all three *classes* of animal in the table below, based on the feature from the table:

Class	**Feature**
Invertebrate	No spine
Osteichthyes	Lives in water
Aves	Flying

Osteichthyes are fish, and Aves are birds.

Your program should ask for the feature then print out the Class of animal that feature corresponds to. For example:

```
What is the feature? No spine
Invertebrate
```

Here is another example:

```
What is the feature? Lives in water
Osteichthyes
```

If the feature isn't one of the three above, your program should print `I'm not sure.` For example:

```
What is the feature? Fur
I'm not sure.
```

Testing

- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing the class for the feature `Flying`.
- ☐ Testing a different feature (`Claws`).
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

4.1.5. Problem: Find the animal!

In Biology, we can use the *characteristics* of an animal to classify them, or separate one from another.

Your teacher has given you a list of Australian animals with a characteristic of each. Write a program to accept an `input` characteristic, and print out the correct animal from the table.

Animal	**Characteristic**
Kangaroo	Pouch
Cassowary	Casque
Goanna	Claws

Your program should ask for the characteristic then print out the corresponding animal. For example:

```
Characteristic? Pouch
Kangaroo!
```

Notice that there is an exclamation mark at the end of the output!

Here is another example:

```
Characteristic? Casque
Cassowary!
```

If the feature isn't one of the three above, your program should print `I do not know.` For example:

```
Characteristic? Hooves
I do not know.
```

Testing

- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing the animal for the characteristic `Claws`.
- ☐ Testing a different characteristic (`Wings`).
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

4.2. Decisions and algorithms

4.2.1. Making complex decisions

As we move further down the classification taxonomy, we often need to ask multiple questions to be able to confidently identify where the animal should be categorised.

Let's say we wanted to separate Aves (Birds) (https://en.wikipedia.org/wiki/Bird), Monotremes (https://en.wikipedia.org/wiki/Monotreme) and Mammals (https://en.wikipedia.org/wiki/Mammal) into separate groups. We'll simplify things a bit and say that the only things we have information about are whether the animal can fly, and if it lays eggs.

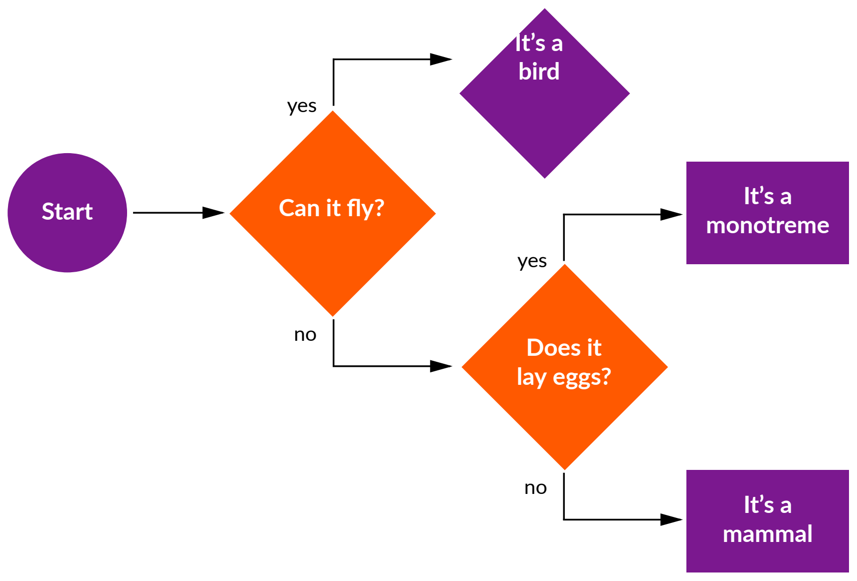
This table summarises some simple facts about these three groups of animals:

	Bird	**Monotreme**	**Mammal**
Can fly?	**Yes**	No	No
Lays eggs?	**Yes**	**Yes**	No

So Birds both lay eggs and can fly, Monotremes lay eggs but don't fly, and mammals don't lay eggs and don't fly.

4.2.2. Asking the right questions

The flowchart below demonstrates how we can use the properties of `can_fly` and `lay_eggs` to determine which animal it is:



Because only birds can fly, we can confidently say that a yes to our first question means we have a bird.

Whether we ask the second question at all *depends* on the answer to the first!

4.2.3. Decisions within decisions

The body of an `if`, `elif` or `else` statement may contain another `if` statement. This is called *nesting*, and is how we represent our algorithm of a decision within a decision, in code.

When using *dichotomous keys* in Biology, our goal is to reduce the number of species until there are no other possibilities.

When we have eliminated all possibilities, we don't need to ask any more questions, so **asking more questions *depends* on the previous answer!**

This is the advantage of using nesting.

In the example from the previous slide, the second decision only happens inside the `'no'` case (or the `else` of our first question). Let's turn the flowchart into code!

```
can_fly = input('Can it fly? ')
if can_fly == 'y':
  print('Aves (Bird)')
else:
  lay_eggs = input('Does it lay eggs? ')
  if lay_eggs == 'y':
    print('Monotreme')
  else:
    print('Mammal')
```

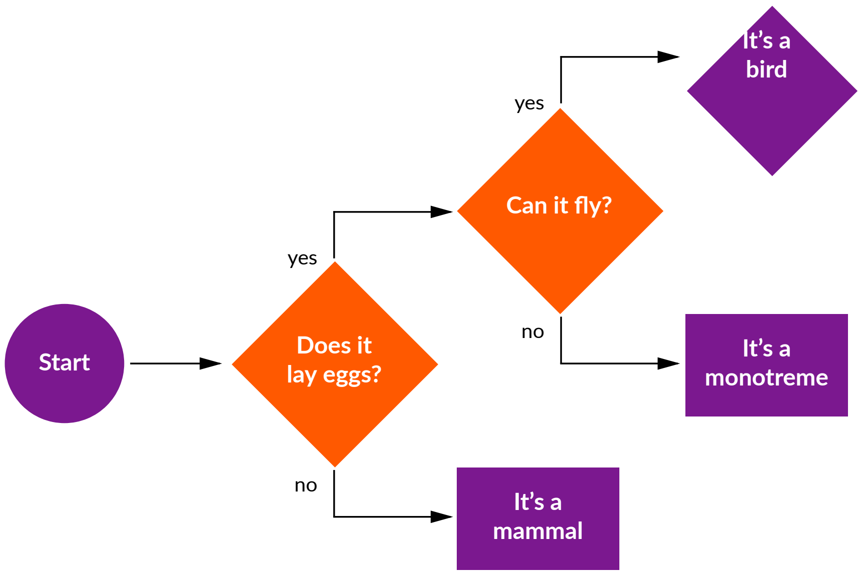
```
Can it fly? n
Does it lay eggs? n
Mammal
```

Notice how the indentation of the `print` and `input` functions looks a bit like the indentation of the rectangles in our flowchart?

Test the example by running the code above. Try different answers to all of the questions. You'll also notice that if you answer `'y'` to the first question, the second question won't get asked at all!

4.2.4. Order matters!

Changing the order that you ask your questions can change your algorithm. Look what happens to the flowchart if we ask if the animal lays eggs first:



This algorithm requires asking the second question - can it fly? - inside the body of the first `if`, not the `else` like we did last time. You can see this by looking at where the second question gets asked in the flowchart - after the first answer is `'yes'`.

This changes how we nest our `if` statements, and changes the code to:

```python
lay_eggs = input('Does it lay eggs? ')
if lay_eggs == 'y':
  can_fly = input('Can it fly? ')
  if can_fly == 'y':
    print('Aves (Bird)')
  else:
    print('Monotreme')
else:
  print('Mammal')
```

Run this program to see how it works, and try different answers to the questions.

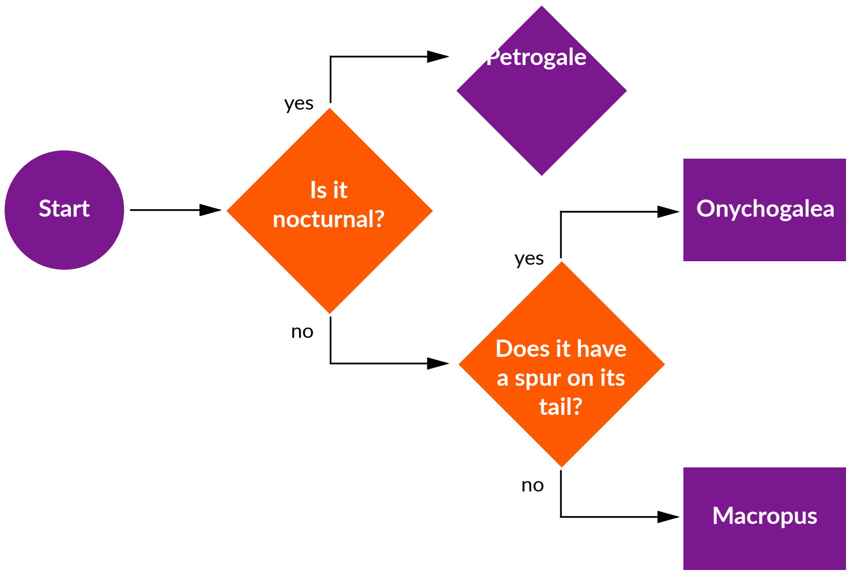
Both algorithms are correct even though the programs are different!

4.2.5. Problem: Which macropod?

There are lots of different types of wallaby, and many of them have distinctive features that place them into different genus groups inside the family Macropodidae (https://en.wikipedia.org/wiki/Macropodidae). Distinctive features of some genuses include:

- Onychogalea (https://en.wikipedia.org/wiki/Nail-tail_wallaby) - Nail-tail wallabies have a horny spur on the end of their tails
- Petrogale (https://en.wikipedia.org/wiki/Rock-wallaby) - Rock-wallabies are very agile. These nocturnal wallabies live in rocky locations where they can find refuge during the day
- Macropus (https://en.wikipedia.org/wiki/Macropus) - The largest of the genuses in this family, many macropods that live on the ground are in this group.

Write a program that will identify the genus according to the algorithm shown below:



The first question is `'Is it nocturnal? '`. if it is, it is from genus `'Petrogale'`:

```
Is it nocturnal? yes
It is from the genus Petrogale.
```

If the answer to the first question is `'no'`, ask `'Does it have a spur on its tail? '` to see if it is `'Onychogalea'`:

```
Is it nocturnal? no
Does it have a spur on its tail? yes
It is from the genus Onychogalea.
```

If the answer to the spur question is `'no'`:

```
Is it nocturnal? no
Does it have a spur on its tail? no
It is from the genus Macropus.
```

Hint!

Remember that the algorithm in the flowchart provides hints about how to nest your `if` statements.

Testing

☐ Testing your input prompt for the first question.

- ☐ Testing the genus Petrogale.
- ☐ Testing your input prompt for the second question.
- ☐ Testing the genus Onychogalea.
- ☐ Testing the genus Macropus.
- ☐ Testing an answer that is not yes or no.
- ☐ Testing a hidden case.

4.2.6. Problem: What's my diet?

In our earlier notes we saw how we could use an `elif` statement to print different messages for animals that were `'carnivore'`, `'herbivore'` or `'omnivore'`. We'll use what we just learned about nesting to do this a little bit differently.

Write a program that asks the user if the animal eats meat or plants and depending on their answer identifies the diet of the animal. It should ask about meat first.

You must ask each question separately and in the correct order, and you should only ask the second question if it is required. Here is how the program behaves when you answer `'no'` to `'Does it eat meat?'`:

```
Does it eat meat? no
It's a herbivore!
```

If you answer `'yes'` to the first question, ask `'Does it eat plants? '`:

```
Does it eat meat? yes
Does it eat plants? no
It's a carnivore!
```

And `'yes'` to both questions:

```
Does it eat meat? yes
Does it eat plants? yes
It's an omnivore!
```

Hint!

If you're having trouble with the nesting, a flowchart might help you work out how to nest your `if` statements.

Testing

- ☐ Testing your input prompt for the first question.
- ☐ Testing a herbivore.
- ☐ Testing your input prompt for the second question.
- ☐ Testing a carnivore.
- ☐ Testing an omnivore.
- ☐ Testing an answer that is not yes or no.
- ☐ Testing a hidden case.

4.3. Decision review

4.3.1. Problem: Nesting or elif?

When is it better to use nesting instead of an `elif`?

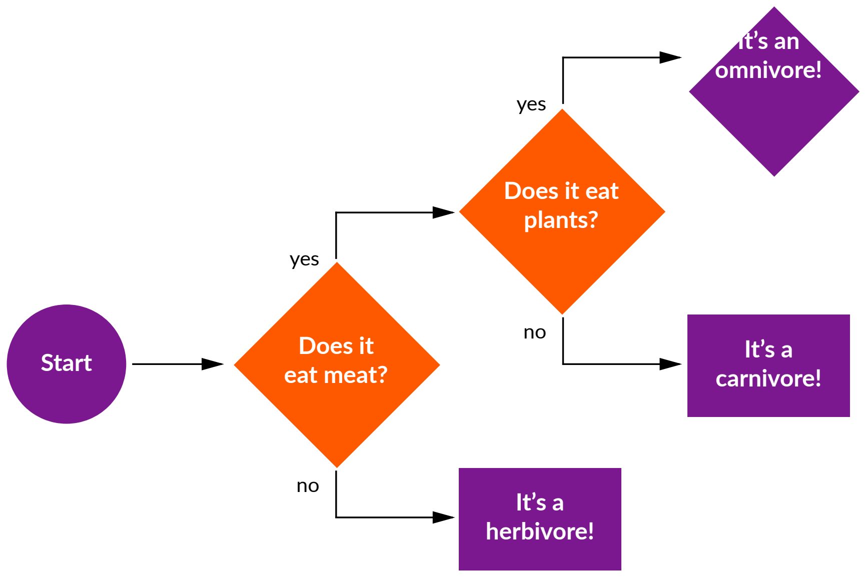
- ○ When there are more than two possible answers to a single question.
- ○ Using `elif` is always the best option.
- ○ When you want to check something that depends on a previous condition.
- ○ Using nesting is always the best option.

Testing

☐ That's right!

4.3.2. Problem: Which is it?

Which of the programs is the correct representation of the algorithm shown in the flowchart below?



○
```
meat = input('Does it eat meat? ')
if meat == 'yes':
  plants = input('Does it eat plants? ')
  if plants == 'yes':
    print("It's a carnivore!")
  else:
    print("It's an omnivore!")
else:
  print("It's a herbivore!")
```

○
```
meat = input('Does it eat meat? ')
if meat == 'no':
  plants = input('Does it eat plants? ')
  if plants == 'yes':
    print("It's a herbivore!")
  else:
    print("It's an omnivore!")
else:
  print("It's a carnivore!")
```

○
```
meat = input('Does it eat meat? ')
if meat == 'yes':
  plants = input('Does it eat plants? ')
  if plants == 'yes':
    print("It's an omnivore!")
  else:
    print("It's a carnivore!")
else:
  print("It's a herbivore!")
```

○
```
meat = input('Does it eat meat? ')
if meat == 'no':
  print("It's a herbivore!")
else:
  plants = input('Does it eat plants? ')
  if plants == 'yes':
    print("It's a carnivore!")
  else:
    print("It's an omnivore!")
```

Testing

☐ That's right!

5

MINI PROJECT: SIMPLE CLASSIFIER

5.1. Project description

5.1.1. What's the project?

So far in the challenge you've learned quite a lot about both Biology and Digital Technologies:

Biology

- *Taxonomy* is the classification of living things
- The *taxonomic rank* describes an organism based on its similarities with others
- The *scientific name* of an organism is a combination of its *Genus* and *Species*
- *Classification* is done based on *features* and characteristics of organisms
- *Dichotomous keys* allow us to split larger groups of animals based on whether a feature is present or not

Digital Technologies

- Computers perform their functions through code *written by people*
- Programming languages are very *precise* and have their own *syntax and grammar*
- We can *display text* on the screen in Python using the `print` function
- The user can *enter text* into our program via the `input` function
- *Variables* are used to store data in a computer program
- An `if` statement is used for *branching* in Python. It runs instructions when a condition is `True`.
- Adding an `elif` or `else` to an `if` statement allows you to specify alternative instructions to run under different conditions.

This project guides you through the process of building your own mini-classifier that uses dichotomous keys to identify an organism based on its features.

5.1.2. Classifying animals

The first step in constructing our classifier is to develop an algorithm that will allow us to correctly identify each of our animals using their features. The animals we will be classifying are:

- Koala
- Frilled-neck lizard
- Murray cod
- Laughing kookaburra

You can download a set of trading cards (https://aca.edu.au/public/resources/classifier-python-mini.pdf) for these animals that gives you the information you need about their features to build a series of dichotomous keys that will classify each animal uniquely.

5.1.3. Problem: Animal features

Using the information provided in the [set of trading cards (https://aca.edu.au/public/resources/classifier-python-mini.pdf)](https://aca.edu.au/public/resources/classifier-python-mini.pdf), identify the features of each animal that will be used for classification.

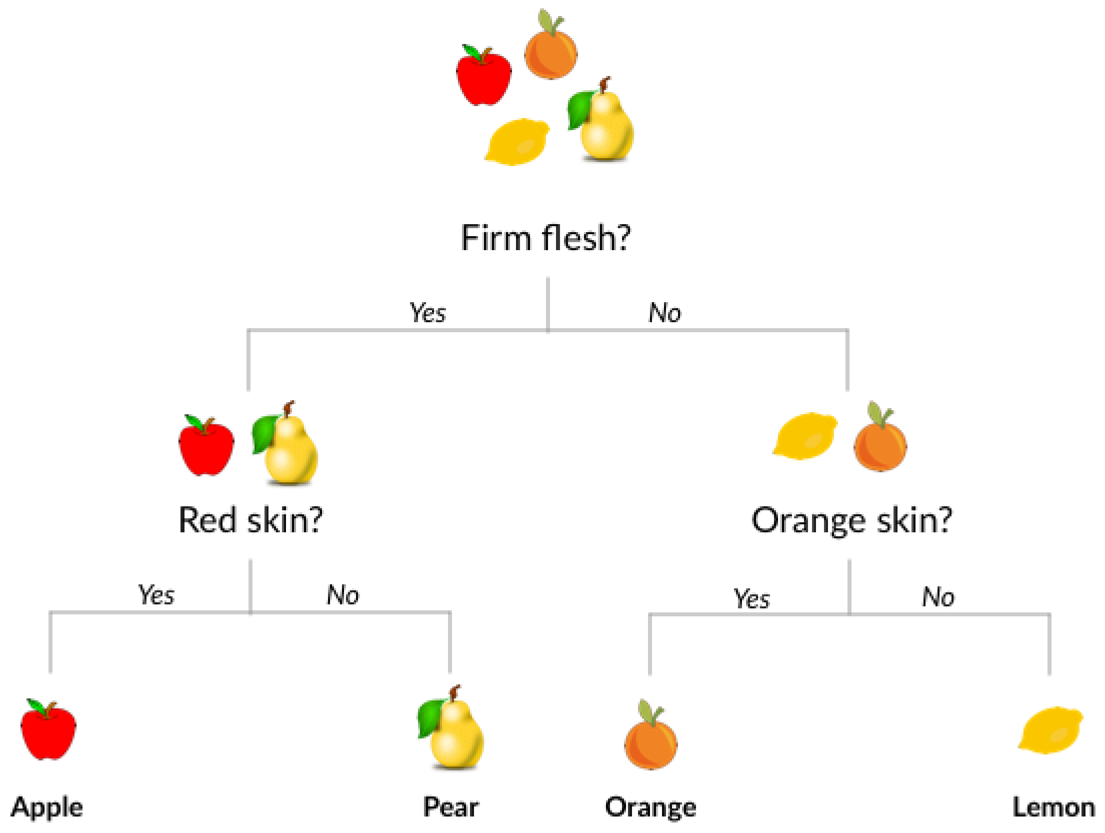
	Fur	Eggs	Claws	Scales
Koala				
Frilled-neck Lizard				
Murray Cod				
Laughing Kookaburra				

Testing

- ☐ Checking the "Koala" row.
- ☐ Checking the "Frilled-neck Lizard" row.
- ☐ Checking the "Murray Cod" row.
- ☐ Checking the "Laughing Kookaburra" row.

5.1.4. Dichotomous tree

Now that you know which features belong to which animal, your next step is to work out an appropriate split so that you can build your dichotomous tree. We used flowcharts to do this in our previous modules, but another common representation of the classification process is shown below:



Classifying fruit using dichotomous keys

This example classifies the four different fruits based on their features using just two questions for each one. You'll be doing a similar thing with our four animals.

[This template (https://aca.edu.au/public/resources/classifier-template.pdf)](https://aca.edu.au/public/resources/classifier-template.pdf) will help you work out which questions you need to ask using the trading cards from the previous slides.

There are a few different solutions to this problem, but the best one will only require asking ***two questions*** to identify each animal uniquely.

You can check your solution with your teacher first, or progress to the next slide and write your program!

5.1.5. Problem: Biology classifier

Now that you've got a dichotomous tree worked out, the final step is to write the code that classifies the animals for us. The program will use features and questions you've identified in your algorithm to print out the scientific and common name of the animal selected.

The program will operate according to the following rules:

- The input prompts specify just the feature being checked, followed by a colon, e.g. `Fur:` or `Eggs:`
- Answers will be `"y"` or `"n"` for all features
- Only features from the trading cards (https://aca.edu.au/public/resources/classifier-python-mini.pdf) are allowed
- The order of the questions does not matter as long as the correct answer is found
- You don't have to use every feature if it is not required
- The same program will work for every animal in the data

One example that correctly identifies a koala might be:

```
Scales: n
Claws: y
Fur: y
The species is Phascolarctos cinereus.
Its common name is koala.
```

It does not matter how many questions or features you use to identify each animal as long as you identify each one *uniquely* and *correctly*. **Challenge yourself and see if you can write a program that only asks two questions!**

Once the animal is identified, you should immediately print its scientific and common names to the screen. You must format the output of your program as shown in the example above.

Use your algorithm

If you've worked through the activities in each of the slides up to this point, you should be able to follow the same process we did in earlier modules to convert your algorithm into code.

Testing

- ☐ Testing the example animal from the question (koala).
- ☐ Testing the laughing kookaburra.
- ☐ Testing the Murray cod.
- ☐ Testing the frilled-neck lizard.

5.2. Project complete

5.2.1. Congratulations!

Excellent work on finishing the project and our biology challenge! You've learnt a little bit about how biological organisms are classified, as well as how to solve some interesting problems with code!

If you want to learn a bit more programming there's a biology extension challenge (https://groklearning.com/learn/aca-dt-7-py-biology-extension/) waiting for you! It teaches you how to work with collections of data called lists, reading from files that contain lots of information, and you'll build a much more complicated classifier that includes extra information about many more animals!