

Welcome to the Labs!

Tic Tac Toe

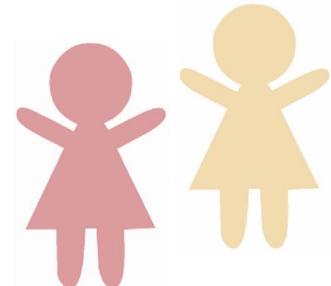


Who are the tutors?

Who are you?

Two Truths and a Lie

1. Get in a group of 3-5 people
2. Tell them three things about yourself:
 - a. Two of these things should be true
 - b. One of these things should be a lie!
3. The other group members have to guess which is the lie



Log on

Log on and jump on the GPN website

girlsprogramming.network/workshop

You can see:

- These **slides** (to take a look back or go on ahead).
- A digital copy of your **workbook**.
- Help bits of text you can **copy and paste!**

There's also links to places where you can do more programming!

Tell us you're here!

Click on the
Start of Day Survey
and fill it in now!

SploitCode

SplootCode is new



We're trying something new!

SplootCode is a bit different to Python

Normal Python

```
print("Hello!")
```

SplootCode Python

```
name = input("Enter name: ")  
print("Hello, " + name)
```

```
f print('Hello!')
```

```
assign v name = f input('Enter name: ')
```

```
f print('Hello, ' + v name )
```

Today's project!

Tic Tac Toe



Using the workbook!

The workbooks will help you put your project together!

Each **Part** of the workbook is made of tasks!

Tasks - The parts of your project

Follow the tasks **in order** to make the project!

Hints - Helpers for your tasks!

Stuck on a task, we might have given you a hint to help you **figure it out!**

The hints have **unrelated** examples, or tips. **Don't copy and paste** in the code, you'll end up with something **CRAZY!**

Task 6.2: Add a blah to your code!

This has instructions on how to do a part of the project

1. Start by doing this part
2. Then you can do this part

Task 6.1: Make the thing do blah!

Make your project do blah

Hint

A clue, an example or some extra information to help you figure out the answer.

```
print('This example is not part of the project' )
```

Using the workbook!

The workbooks will help you put your project together!

Check off before you move on from a **Part**! Do some bonuses while you wait!

Checklist - Am I done yet?

Make sure you can tick off every box in this section before you go to the next Part.

Lecture Markers

This tells you you'll find out how to do things for this section during the names lecture.

Bonus Activities

Stuck waiting at a lecture marker?
Try a purple bonus. They add extra functionality to your project along the way.

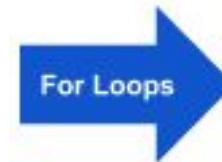


CHECKPOINT



If you can tick all of these off you're ready to move the next part!

- Your program does blah
- Your program does blob



★ BONUS 4.3: Do some extra!

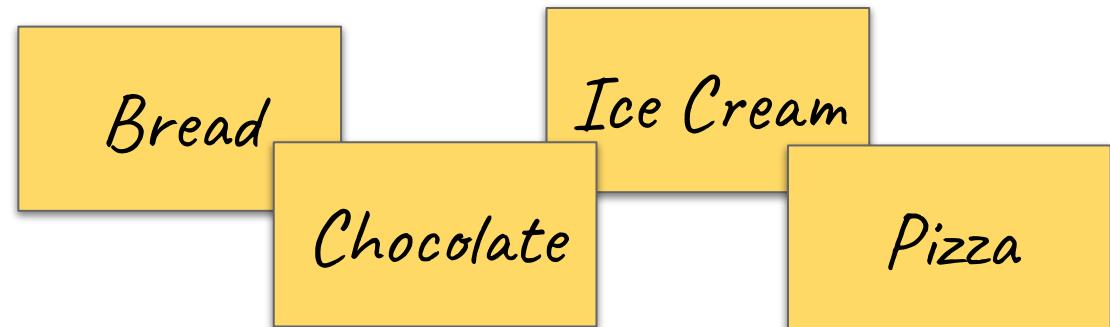
Something to try if you have spare time before the next lecture!

Lists

Lists

When we go shopping, we write down what we want to buy!

But we don't store it on lots of little pieces of paper!



We put it in one big shopping list!

-
- The diagram shows a single large yellow rectangular box containing a bulleted list. The list items are: '● Bread', '● Chocolate', '● Ice Cream', and '● Pizza'. Each item is preceded by a solid black circle.
- Bread
 - Chocolate
 - Ice Cream
 - Pizza

Lists

It would be annoying to store it separately when we code too

```
assign v shopping_item_1 = 'Bread'  
assign v shopping_item_2 = 'Chocolate'  
assign v shopping_item_3 = 'Ice Cream'
```

So much repetition!

Instead we use a python list!

```
assign v shopping_list = list {'Bread'  
                                {'Chocolate'  
                                {'Ice Cream'}}
```



You can put (almost) anything into a list

You can have a list of
integers

```
assign v primes = list { 2 }  
                        { 3 }  
                        { 5 }  
                        { 7 }
```

You can have **lists** with mixed
integers and **strings**

```
assign v mixture = list { 1 }  
                        { 'two' }  
                        { 3 }  
                        { 4 }  
                        { 'five' }
```

But this is almost never a
good idea! You should be
able to treat every element of
the **list** the same way.

List anatomy

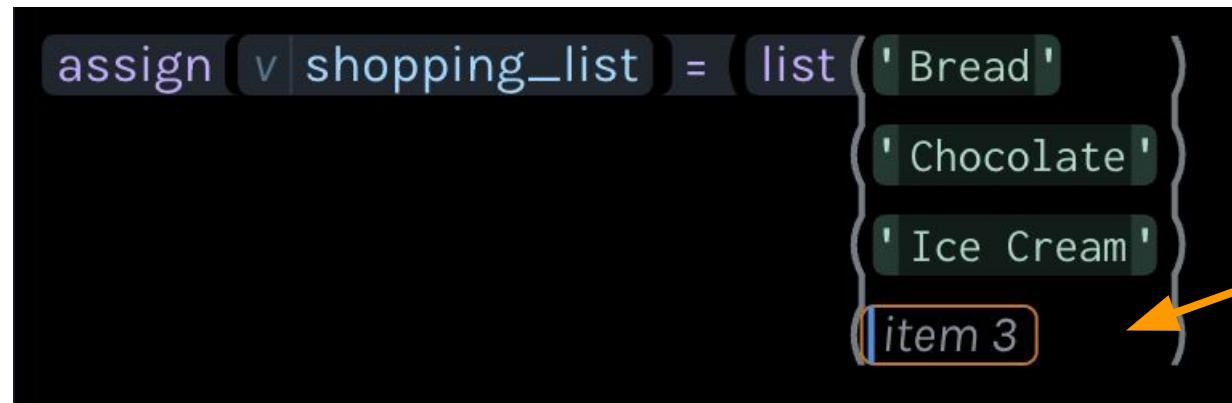
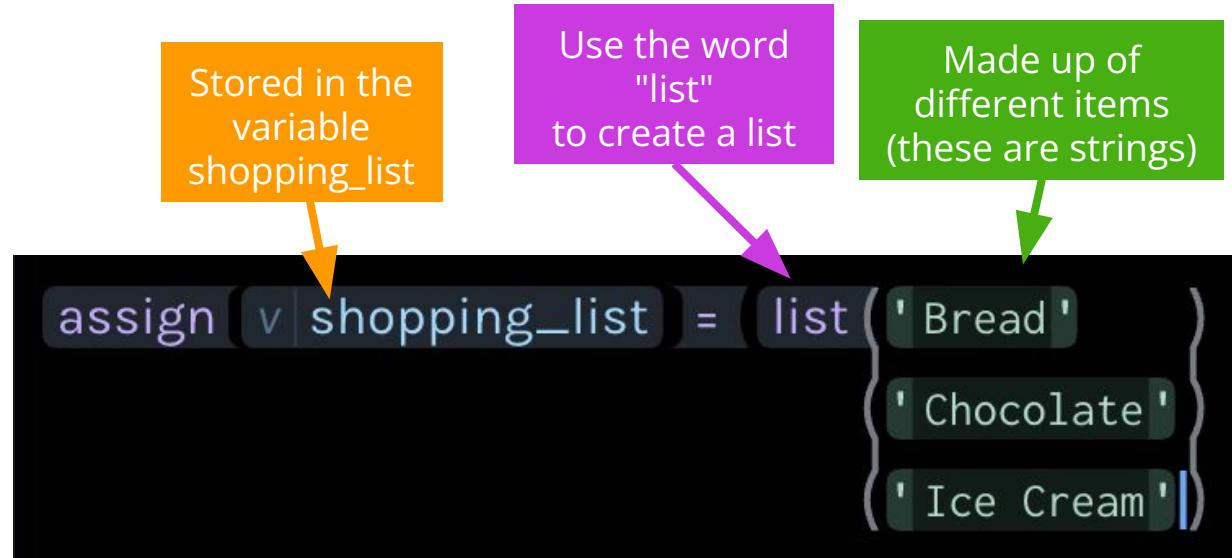
The diagram shows a code snippet for creating a list:

```
assign v shopping_list = list {'Bread'}  
{'Chocolate'}  
{'Ice Cream'}
```

Annotations explain the components of the list:

- An orange box points to the variable assignment: "Stored in the variable shopping_list".
- A purple box points to the word "list": "Use the word \"list\" to create a list".
- A green box points to the list items: "Made up of different items (these are strings)".

List anatomy



Accessing Lists!

The favourites **list** below holds four strings in order.

```
assign ( v faves ) = ( list { ' books '
                                ' butterfly '
                                ' chocolate '
                                ' skateboard ' } )
```

We can count out the items using index numbers!

0



1



2



3



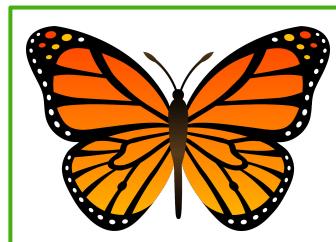
Remember: Indices start from zero!

Accessing Lists

We access the items in a **list** with an index such as 0:

```
assign v faves = list ('books'  
                      {'butterfly'}  
                      {'chocolate'}  
                      {'skateboard'})  
  
v faves item (0) → "books" (str)
```

What code do you need to access the second item in the list?



Accessing Lists

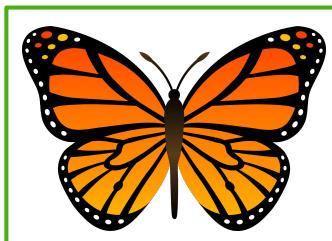
We access the items in a **list** with an index such as 0:

```
assign v faves = list ('books'  
                      {'butterfly'}  
                      {'chocolate'}  
                      {'skateboard'})  
  
v faves item (0) → "books" (str)
```

What code do you need to access the second item in the list?

0

```
v faves item (1)
```



2



3

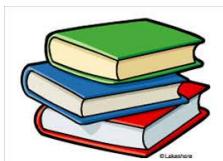


Going Negative

Negative indices count backwards from the end of the **list**:

```
v faves item(-1) → "skateboard" (str)
```

What would `v faves item(-2)` return?



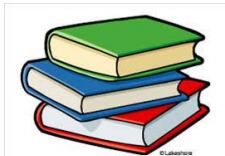
Going Negative

Negative indices count backwards from the end of the **list**:

```
v faves item(-1) → "skateboard" (str)
```

What would `v faves item(-2)` return?

-4



-3



-2



-1



Falling off the edge

Python complains if you try to go past the end of a **list**

```
assign ( v | faves ) = ( list ( ' books ' ) ) → [ 'books', 'butterfly', 'chocolate', 'skateboard' ] (list)
```

```
{ ' butterfly ' }  
 { ' chocolate ' }  
 { ' skateboard ' }
```

```
v | faves | item ( 4 ) IndexError: list index out of range
```

Updating items!

We can also update things in a list:

```
v faves item( 2 ) → "chocolate" (str)  
assign v faves item( 2 ) = ' lollipops ' → "lollipops" (str)
```



Updating items!

We can also update things in a list:

```
v | faves | item( 2 ) → "chocolate" (str)
```

```
assign( v | faves | item( 2 ) ) = ' lollipops ' → "lollipops" (str)
```

```
v | faves | item( 2 ) → "lollipops" (str)
```

```
v | faves → ['books', 'butterfly', 'lollipops', 'skateboard'] (list)
```



List of lists!

You really can put anything in a list, even more lists!

We could use a list of lists to store different sports teams!

```
assign v tennis_pairs = list { list { 'Alex' } }  
                                { 'Emily' }  
                                { list { 'Kass' } }  
                                { 'Annie' }  
                                { list { 'Amara' } }  
                                { 'Viv' }
```

Get the first pair in the list

```
assign v first_pair = v tennis_pairs item ( 0 ) → ['Alex', 'Emily'] (list)
```

Now we have the first pair handy, we can get the first the first player of the first pair

```
v first_pair item ( 0 ) → "Alex" (str)
```

Project time!

You now know all about lists!

Let's put what we learnt into our project

Try to do the next Part

The tutors will be around to help!

Functions!

Simpler, less repetition, easier to read code!



How functions fit together!

Functions are like factories!

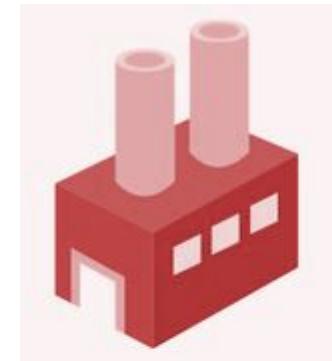
Timber Mill



Your main factory!



Metal Worker

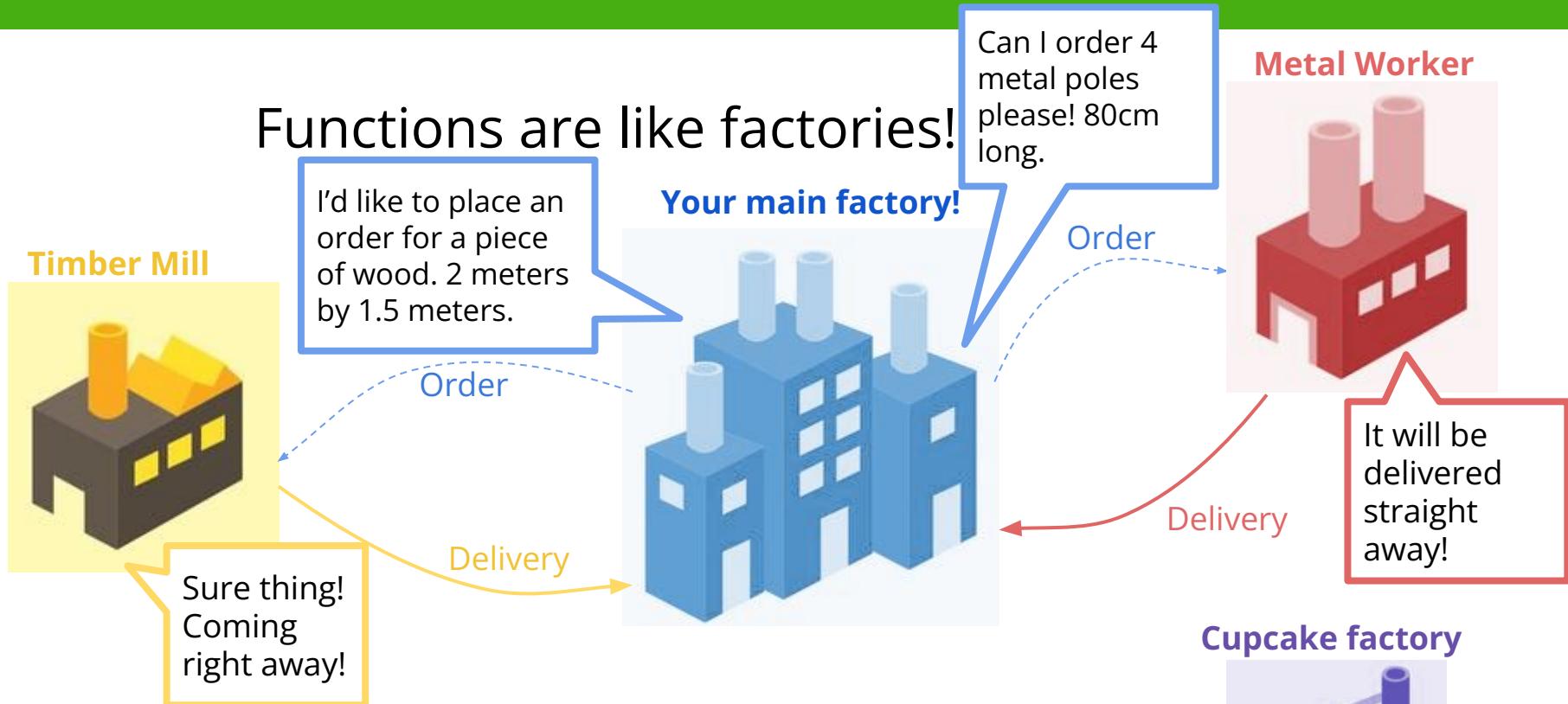


Cupcake factory



Running a factory doesn't mean doing all the work yourself, you can get other factories to help you out!

How functions fit together!



Asking other factories to do some work for you makes your main task simpler. You can focus on the assembly!



How functions fit together!

Functions are like factories!

Timber Mill

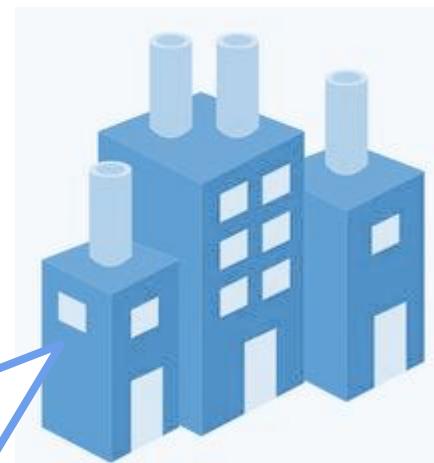


Look at this beautiful table I made!

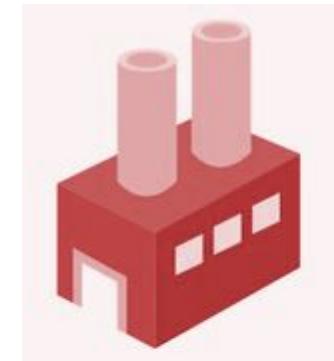


Outsourcing made it simple!

Your main factory!



Metal Worker

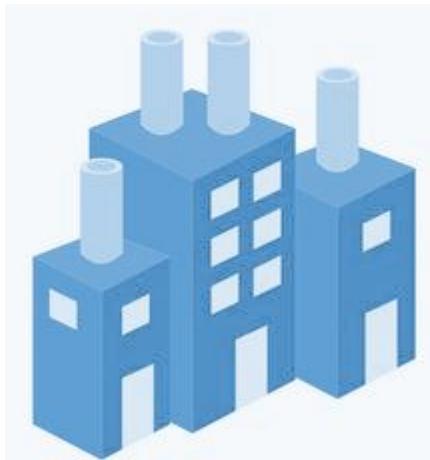


Cupcake factory



How functions fit together!

Your main code!



You can write a bunch of helpful functions to **simplify** your **main goal**!

You can **write** these **once** and then **use** them **lots** of times! They can be **anything** you like!

Uses stats to make decisions



Helps with printing nicely



Does calculations



Don't reinvent the wheel

We're already familiar with some python built-in functions like print and input!

There's lots of functions python gives us to save us reinventing the wheel!

For instance we can use len to get the length of a string, rather than having to write code to count every letter!

```
f | len ('Hello world!') → 12 (int)
```

Try these:

```
assign v name = 'Renee'  
f | len ( v name ) → 5 (int)
```

```
f | int ('6') → 6 (int)
```

```
f | str ( 6 ) → "6" (str)
```



Defining your own functions

Built in functions are great! But sometimes we want custom functions!

Defining our own functions means:

- We cut down on repeated code
- Nice function names makes our code clear and easy to read
- We can move bulky code out of the way



Defining your own functions

```
function v|cat_print() {
    f|print('
#   '
#
#   '
^..^   #
=TT= #####;
#####
# #   # #
M M   M M
```



Defining your own functions

Then you can use your function by calling it!

```
Called 2 times < 1 >
function v cat_print()
    f print(''') # #
    ^..^ # #
    =TT= #####;
    ##########
    # # # #
    M M M M

f cat_print() → None
f cat_print() → None
```

Which will do this!

^..^ #
=TT= #####;

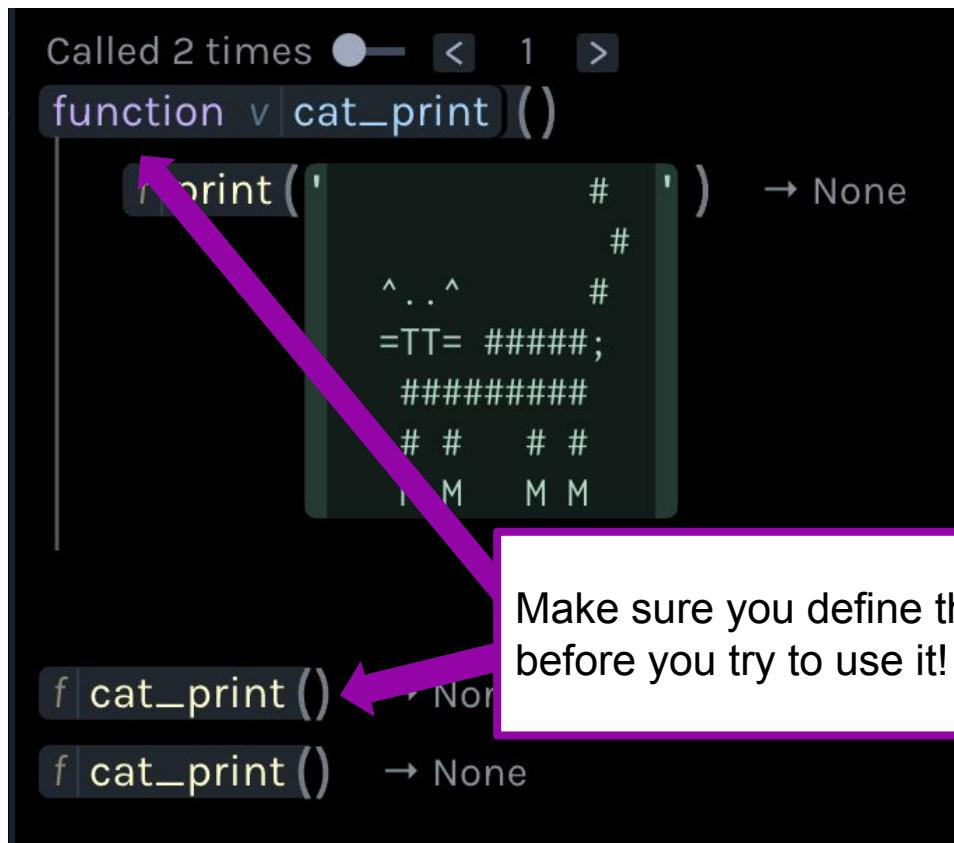
M M M M

^..^ #
=TT= #####;

M M M M

Defining your own functions

Then you can use your function by calling it!



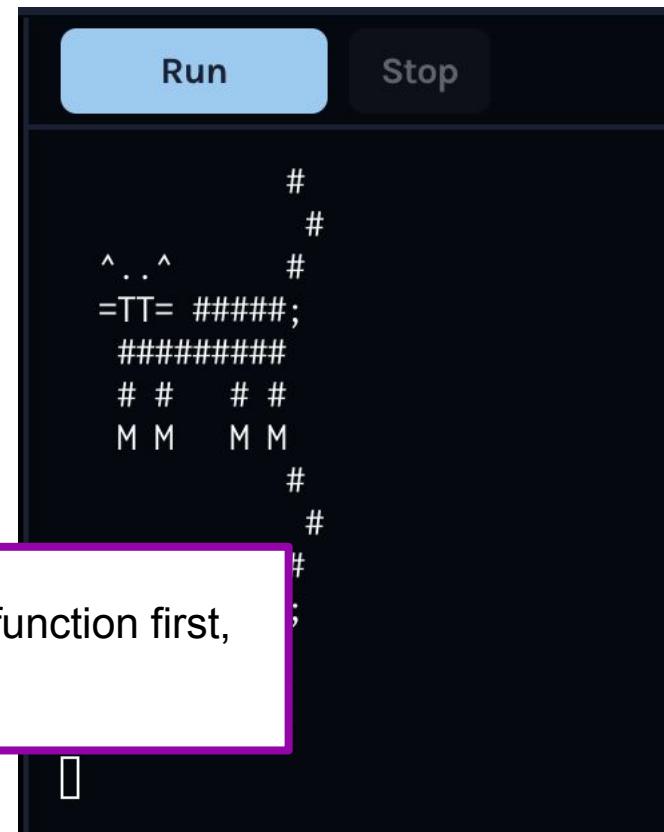
A Scratch script window showing the creation and use of a custom function. The script defines a function named 'cat_print()' which prints a cat pattern to the stage. The function code is:

```
function v cat_print()()
    print( ' # # # # # '
          ' ^..^ # '
          '=TT= #####; '
          ##### #####
          '# #' # #
          M M M M
    ) → None
```

The function is defined at the top of the script. Below it, two instances of the function are used, both resulting in the output 'None'.

Make sure you define the function first, before you try to use it!

Which will do this!



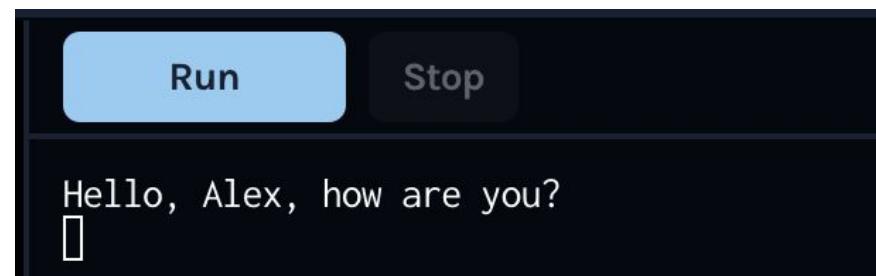
Functions often need extra information

Functions are more useful if we can change what they do
We can do this by giving them arguments (aka parameters)

```
Called 1 times
function v hello( v person )
    f print('Hello, ' + v person + ', how are you?') → None

f hello('Alex') → None
```

Here, we give the hello() function a name



A screenshot of a Python code editor. At the top, there are two buttons: "Run" (highlighted in blue) and "Stop". Below the buttons, the code "Hello, Alex, how are you?" is displayed in a text area. The "Run" button is highlighted in blue, indicating the code has been executed.



Functions can take multiple arguments

Often we want to work with multiple pieces of information.

You can actually have as many parameters as you like!

This function takes two numbers, adds them together and prints the result.

```
Called 1 times
function v add ( v x v y )
|   f print( v x + v y ) → None

f add { 3 }
{ 4 } → None
```



Arguments stay inside the function

The arguments are not able to be accessed outside of the function declaration.

```
Called 1 times
function v|hello| ( v|person| )
    f| print('Hello, ' + v|person| + ', how are you?') → None
f|hello('Alex') → None
v| person| NameError: name 'person' is not defined
```



Variables stay inside the function

Neither are variables made inside the function. They are **local variables**.

```
Called 1 times
function v|add(v|x v|y)
    assign(v|result) = (v|x + v|y) → 7(int)
    f|print(v|result) → None

f|add{3} → None
{4}

v|result NameError: name 'result' is not defined
```



Global variables are not affected

Changing a variable in a function **only changes it *inside* the function.**

```
assign v|result = 1 → 1(int)
Called 1 times
function v|add( v|x v|y )
    assign v|result = v|x + v|y → 7(int)
    f|print( v|result ) → None
f|add { 3 }
      { 4 }
v|result → 1(int)
```



Recap: A function signature

the **function** keyword

function **name**

function **arguments**

definition

```
function v add( v x v y )
```

calling a function

function **name**

function **arguments**

Giving something back

At the moment our function just does a thing, but it's not able to give anything back to the main program.

Currently, we can't use the result of add()

```
Called 1 times
function v|add( v|x v|y )
    assign v|result = v|x + v|y → 5 (int)
    f|print(v|result) → None

assign v|sum = f|add{2
                    3} → None
v|sum → None
```

sum has no value!



Giving something back

Using `return` in a function returns a result.

```
Called 1 times
function v|add( v|x v|y )
    assign v|result = ( v|x + v|y ) → 5 (int)
    return v|result → 5 (int)

assign v|sum = ( f|add { 2 }
                  { 3 } ) → 5 (int)
v|sum → 5 (int)
```



Giving something back

When a function returns something, the *control* is passed back to the main program immediate, so no code after the `return` statement is run.

```
Called 1 times
function v| add ( v|x v|y )
    assign v|z = v|x + v|y → 5 (int)
    f| print ('Before return') → None
    return v|z → 5 (int)
    f| print ('After return') ←

assign v|sum = f|add { 2 }
                  { 3 } → 5 (int)
v|sum → 5 (int)
```

Here, the `print` statement after the `return` never gets run.



Project time!

Now go be functional.

Do the next part of the project!

Try to do Part 3

The tutors will be around to help!



If Statements

Conditions

So to know whether to do something, they find out if it's **True!**

```
assign v fave_num = 5 → 5 (int)
if v fave_num < 10 → True
  f print('That's a small number') → None
```

Else statements

else
statements
means something
still happens if
the **if** statement
was **False**

```
assign v word = 'Chocolate'  
  
if v word == 'GPN'  
    f print(' GPN is awesome! ')  
  
else  
    f print(' The word isn't GPN :( )')
```

What happens?

Else statements

else
statements
means something
still happens if
the **if** statement
was **False**

```
assign v word = 'Chocolate' → "Chocolate" (str)
if v word == 'GPN' → False
  f print ('GPN is awesome!')
else
  f print ('The word isn't GPN :(') → None
```

What happens?

The word isn't GPN :(



Elif statements

else if
means another
if is checked if
the first **if** was
False

```
assign v word = 'Chocolate'  
if v word == 'GPN'  
    f print(' GPN is awesome! ')  
else if v word == 'Chocolate'  
    f print(' YUMMMMM! ')  
else  
    f print(' The word isn't GPN :( )')
```

What happens?

Elif statements

else if
means another
if is checked if
the first **if** was
False

```
assign v word = 'Chocolate' → "Chocolate" (str)
if v word == ' GPN ' → False
    f print(' GPN is awesome! ')
else if v word == ' Chocolate ' → True
    f print(' YUMMMMM! ') → None
else
    f print(' The word isn't GPN :( )')
```

What happens?

YUMMMMM!

Booleans (True and False)

Python has some special comparisons for checking if something is **in** something else. **Try these!**

'A'	in	'AEIOU'	→ True
'Z'	in	'AEIOU'	→ False
'a'	in	'AEIOU'	→ False

```
assign v animals = list{'cat'}  
{'dog'}  
{'goat'}  
  
'banana' in v animals → False  
  
'cat' in v animals → True
```

Project Time!

You now know all about **if!**

See if you can do the next Part

The tutors will be around to help!

While Loops

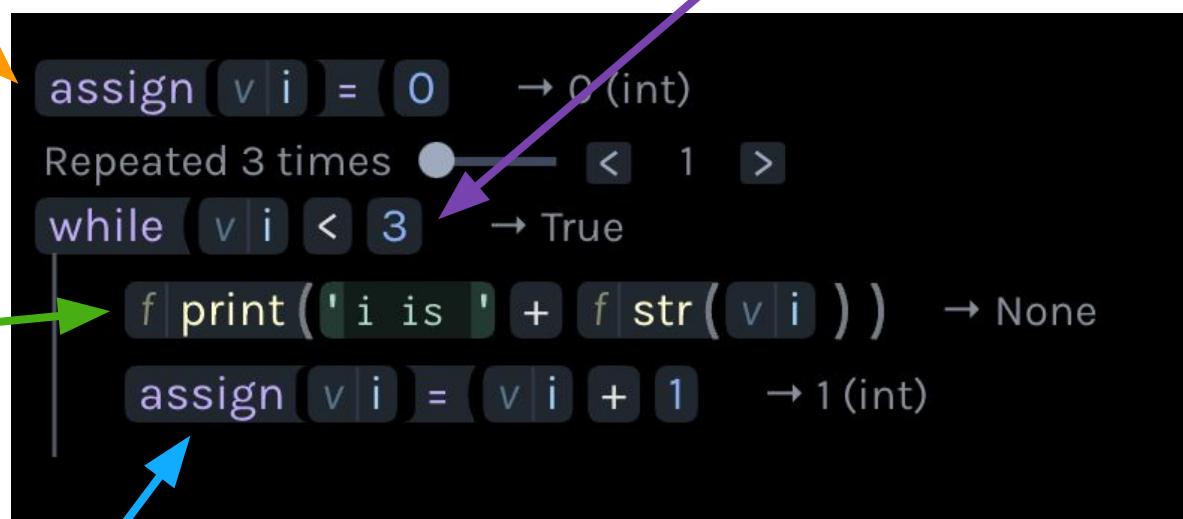
Introducing ... while loops!

Initialise the loop variable

Loop condition

Code to repeat

Update the loop variable



A Scratch script illustrating a while loop. It starts with an 'assign' block setting a variable *v|i* to 0, followed by a note '→ 0 (int)'. Below it is a 'while' loop condition: 'while (v|i < 3) → True'. Inside the loop, there is a 'repeat' block containing a 'print' command: 'print ("i is " + str(v|i)) → None'. After the repeat block, another 'assign' block increments the variable *v|i* by 1, resulting in 1 (int). A green arrow points from the 'Code to repeat' callout to the 'repeat' block. An orange arrow points from the 'Initialise the loop variable' callout to the first 'assign' block. A purple arrow points from the 'Loop condition' callout to the 'while' loop condition.

i is 0
i is 1
i is 2

What happens when.....

What happens if we forget to update the loop variable?

```
assign v i = 0
while v i < 3
    f print(' i is ' + f str(v i) )
```

Give me a break!

But what if I wanna get out of a loop early?

That's when we use the **break** keyword!

```
assign v number = 0

while v number ≠ 42
    assign v number = f input('Guess a number: ')
    if v number == 'I give up'
        f print('The number was 42')
        break

    assign v number = f int(v number)
```

```
Guess a number: 5
Guess a number: 9
Guess a number: 45
Guess a number: 32
Guess a number: 80
Guess a number: I give up
The number was 42
```



Continuing on

How about if I wanna skip the rest of the loop body and loop again? We use **continue** for that!

```
assign v|number | = 0

while v|number | ≠ 42
    assign v|number | = f|input('Guess a number: ')
    if not v|number |.isnumeric()
        f|print('That's not a number. Try again.')
        continue

    assign v|number | = f|int(v|number |)
```

```
Guess a number: 5
Guess a number: five
That's not a number. Try again.
Guess a number: I give up
That's not a number. Try again.
Guess a number: 45
Guess a number: 42
```



Project Time!

while we're here:

Try to do the next Parts!

The tutors will be around to help!

For Loops



Looping through lists!

What would we do if we wanted to print out this list, one word at a time?

```
assign v words = list{'This'
                      {'is'}
                      {'a'}
                      {'sentence'}

f print(v words item(0))
f print(v words item(1))
f print(v words item(2))
f print(v words item(3))
```

What if it had a 100 items??? That would be **BORING!**



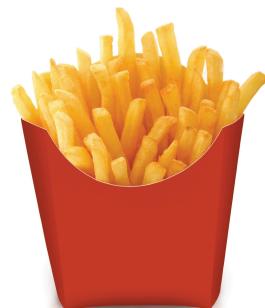
For Loops

For loops allow you to do something for **each** item in a **group** of things

There are many real world examples, like:



**For each page in this book:
Read page**



**For each chip in this bag of chips:
Eat chip**

Looping over a list of ints

We can loop through a list:

```
assign(v|words) = list{'This'  
                      {'is'  
                      {'a'  
                      'sentence'}}
```



```
for(v|word) in(v|words)  
    f|print(v|word)
```

What's going to happen?



Looping over a list of ints

We can loop through a list:

```
assign(v|words) = list{'This'  
                      {'is'  
                      {'a'  
                      'sentence'}}
```



```
for(v|word) in(v|words)  
    f print(v|word)
```

- Each item of the list takes a turn at being the variable **word**
- Do the body once for each item
- We're done when we run out of items!

```
This  
is  
a  
sentence
```



Project Time!

Now you know how to use a for loop!

Try to do Part 5
...if you are up **for it!**

The tutors will be around to help!



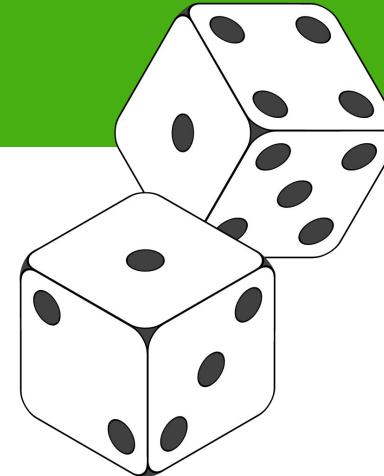
Random!

That's so random!

**There's lots of things in life that
are up to chance or random!**



Python lets us **import** common
bits of code people use! We're
going to use the **random** module!



**We want the computer to
be random sometimes!**



Using the random module

Let's choose something randomly from a list!

This is like drawing something out of a hat in a raffle!

1. Import the random module!

```
import random
```

2. Make a shopping list:

```
assign v shopping_list = list ('Bread'  
                             'Chocolate'  
                             'Ice Cream'  
                             'Pizza')
```



3. Choose randomly! Try it a few times!

```
v random).choice(v shopping_list)
```

Using the random module

You can also assign your random choice to a variable

```
import random

assign v|shopping_list = list {'Bread'
                               'Chocolate'
                               'Ice Cream'
                               'Pizza'} → ['Bread', 'Chocolate', 'Ice Cream', 'Pizza'] (list)

assign v|random_food = v|random|.choice(v|shopping_list) → "Ice Cream" (str)
v|random_food → "Ice Cream" (str)
```



Project Time!

Raaaaaaaaandom! Can you handle that?

Let's try use it in our project!

Try to do the next Part

The tutors will be around to help!

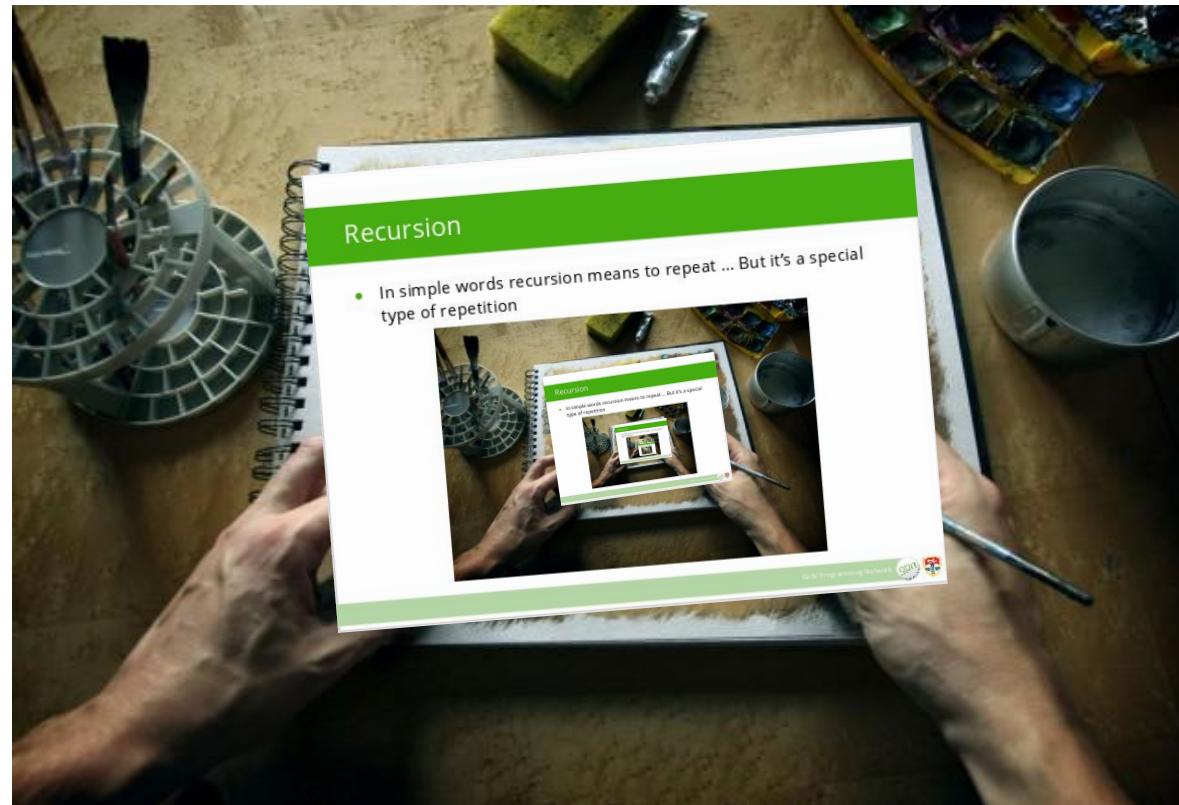
Recursion

Outline

1. What is recursion
2. Recursive function

Recursion

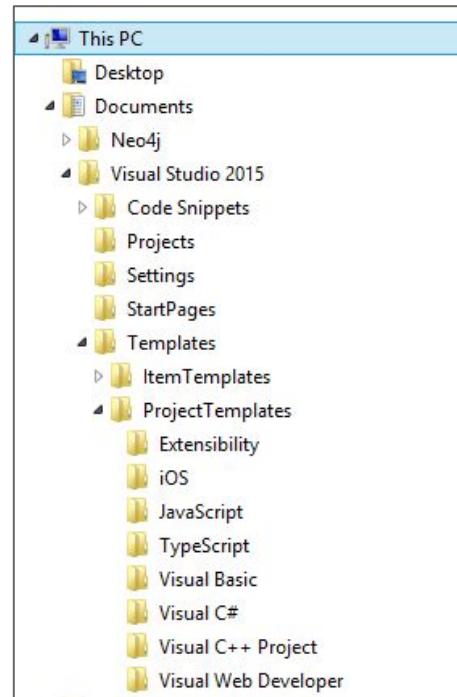
- In simple words recursion means to repeat ... But it's a special type of repetition



Examples

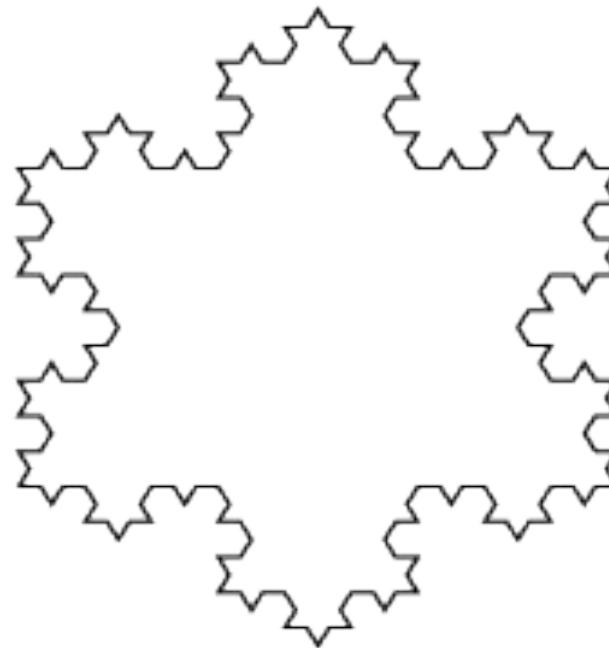
Recursion means “defining something in terms of itself” usually at some smaller scale, perhaps multiple times, to achieve your objective.

“A folder is a structure that holds files and (smaller) folders”



Drawing a snowflake

- How could we draw a snowflake using recursion?



Drawing a snowflake

1

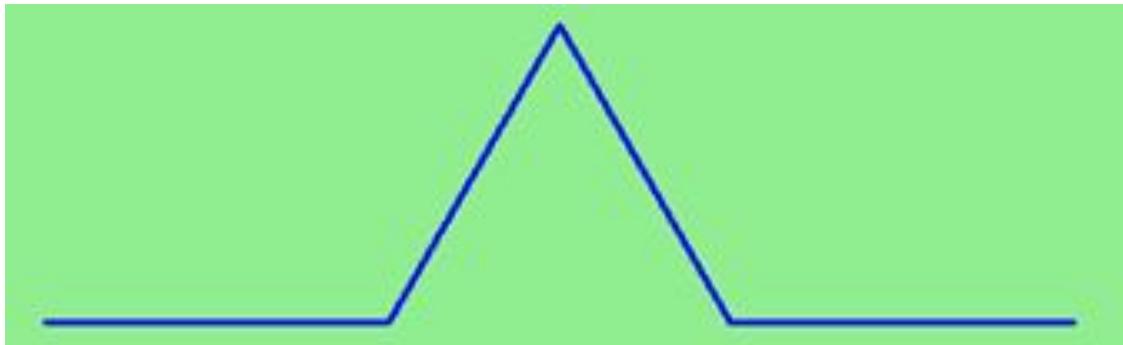
Let's start by finding the simplest shape in the snowflake:
a straight line



Drawing a snowflake

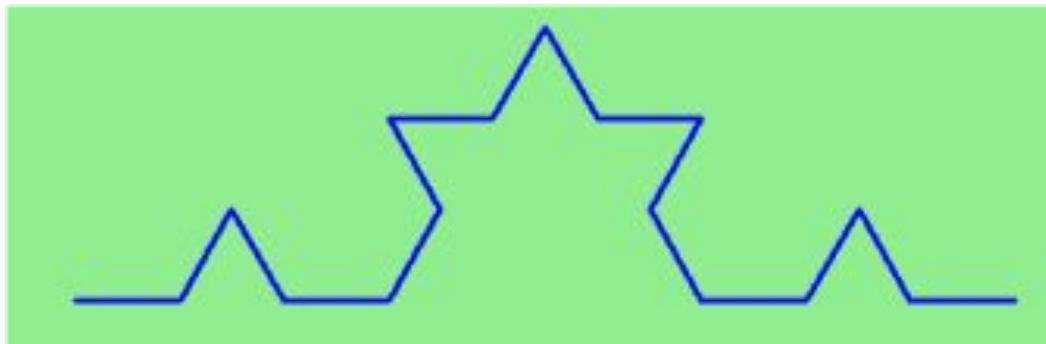
2

Then, we create a bump in the middle of the straight line



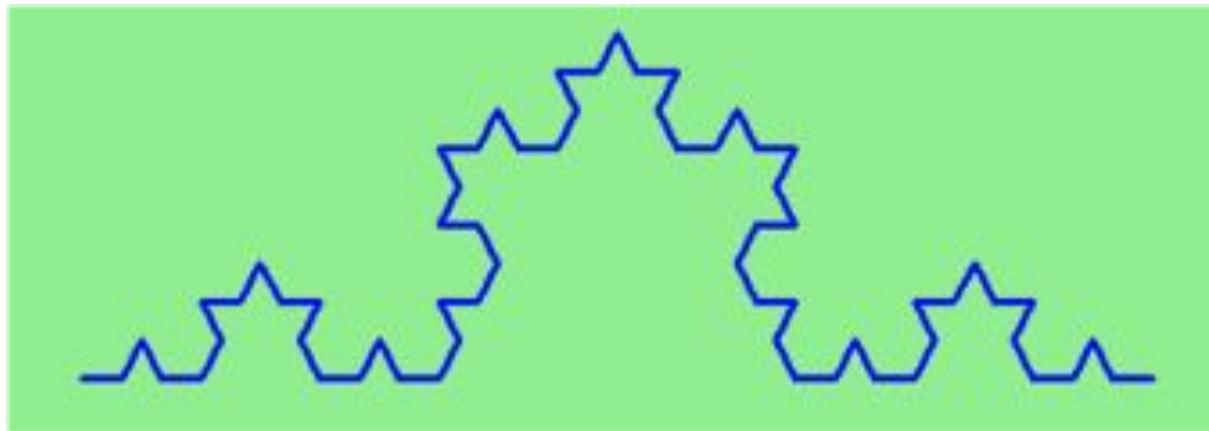
Drawing a snowflake

- What would happen if we repeated steps **1** and **2**?
 - (1) Find straight lines
 - (2) Create a bump in the middle of the straight line



Drawing a snowflake

- Repeat steps **1** and **2** again?
 - (1) Find straight lines
 - (2) Create a bump in the middle of the straight line



What is a *recursive* function?

1. **A recursive function calls itself** with a slightly different argument each time forming layers of repeated function calls that each produce their own result.
2. **There is an end-case** that breaks the recursion and brings you back to the first layer, producing one final result.

Code example

Let's imagine we want to write a function that works out if a word is a Palindrome

anna banana kayak rotator water gpn

Code example

Let's imagine we want to write a function that works out if a word is a Palindrome

anna banana kayak rotator water gpn

A Palindrome is a word that is the same forwards and backwards!

Code example

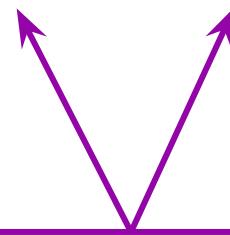
How would you work this out by hand? Let's use this word as an example:

rotator

Code example

How would you work this out by hand? Let's use this word as an example:

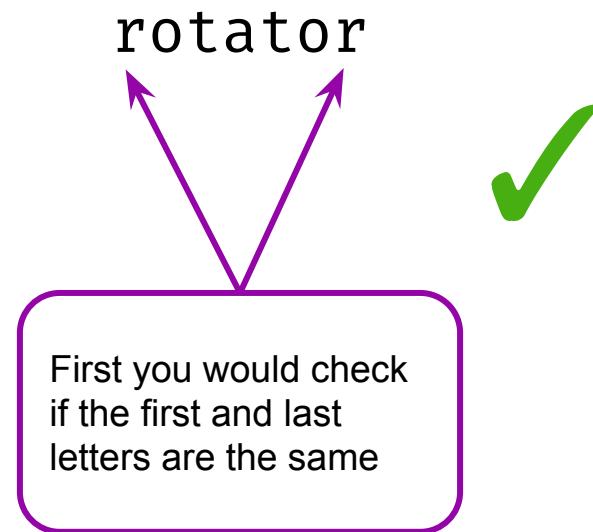
rotator



First you would check
if the first and last
letters are the same

Code example

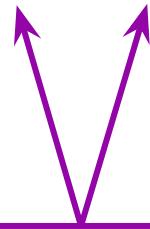
How would you work this out by hand? Let's use this word as an example:



Code example

How would you work this out by hand? Let's use this word as an example:

rotator

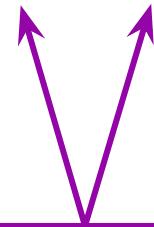


Then you would check
the next two...

Code example

How would you work this out by hand? Let's use this word as an example:

rotator



Then you would check
the next two...

Code example

How would you work this out by hand? Let's use this word as an example:

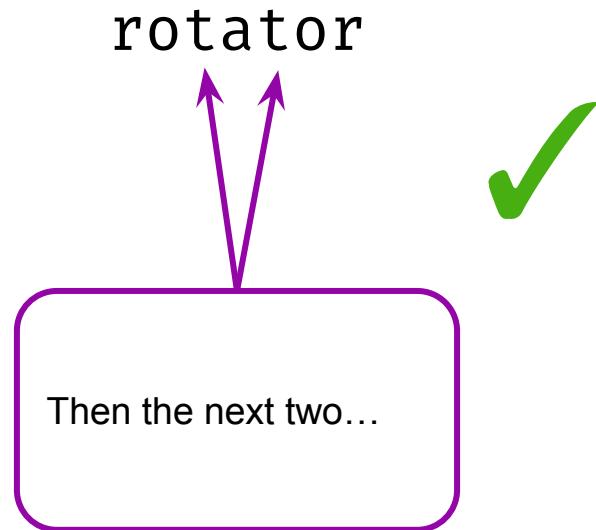
rotator



Then the next two...

Code example

How would you work this out by hand? Let's use this word as an example:



Code example

How would you work this out by hand? Let's use this word as an example:

rotator



Then we only have 1
left, so we're done!

Code example

How would you work this out by hand? Let's use this word as an example:

rotator

So how would you write this in code? 🤔

Code example

How would you work this out by hand? Let's use this word as an example:

rotator

So how would you write this in code? 🤔

You *could* use a loop to figure this out, but we're going to use ✨**recursion**✨

Code example

How would you work this out by hand? Let's use this word as an example:

rotator

The best way to start with recursion is to think about when you want to STOP.

When did we stop checking if the word was a palindrome?

Code example

How would you work this out by hand? Let's use this word as an example:

rotator

The best way to start with recursion is to think about when you want to STOP.

When did we stop checking if the word was a palindrome?

When we got to just one letter!

Code example

Here is what we call our “base case” - this means that this is the smallest problem our function can solve.

```
function v palindrome( v word )
```

```
    if ( f len( v word ) < 2
```

```
        return ( True
```

Code example

Here is what we call our “base case” - this means that this is the smallest problem our function can solve.

E.g. the word “t” is a palindrome (and this function knows it!)

```
Called 2 times ⏪ < 1 >
function v| palindrome( v| word )
    if ( f| len( v| word ) < 2 → True
        return True → True

f| palindrome( '' ) → True
f| palindrome( 't' ) → True
```

Next we need to think about the next size up for our problem.

Code example

Now we are checking the letters on the outside of the word. What do we do next? (hint: this is where the recursion happens)

```
function v|palindrome)( v|word )
    if f|len( v|word ) < 2
        return True
    if v|word|item( 0 ) == v|word|item( -1 )
```

Code example

Now we are checking the letters on the outside of the word. What do we do next? (hint: this is where the recursion happens)

Called 1 times

```
function v| palindrome ( v| word )  
    v| word → "mum" (str)  
    if f| len ( v| word ) < 2 → False  
        return True  
  
    if v| word | item ( 0 ) == v| word | item ( -1 ) → True  
        assign v| middle = ( v| word ) slice ( 1 : -1 ) → "u" (str)  
        #| What next??  
  
    return False → False
```

Code example

```
Called 2 times ●— < 1 >
function v| palindrome( v| word )
    v| word → "mum" (str)
    if f| len( v| word ) < 2 → False
        return True
    if v| word item( 0 ) == v| word item( -1 ) → True
        assign v| middle = v| word slice( 1 : -1 ) → "u" (str)
        return f| palindrome( v| middle ) → True
    return False

f| palindrome( ' mum' ) → True
```

Code example

```
Called 4 times ━━━━ < 1 >
function v palindrome ( v word )
    v word → "rotator" (str)
    if f len ( v word ) < 2 → False
        return True
    if v word item ( 0 ) == v word item ( -1 ) → True
        assign v middle = v word slice ( 1 : - 1 ) → "otato" (str)
        return f palindrome ( v middle ) → True
    return False
```

word = "rotator"



Code example

Called 4 times ●— < 1 >
function v palindrome (v word)

v word → "rotator" (str)

if f len (v word) < 2 → False

return True

False

word = "rotator"

if v word item (0) == v word item (-1) → True

assign v middle = v word slice (1 : - 1) → "otato" (str)

return f palindrome (v middle) → True

return False

Code example

Called 4 times ●— < 1 >
function v palindrome (v word)

v word → "rotator" (str)

if f len (v word) < 2 → False

return True

if v word item (0) == v word item (-1) → True

assign v middle = v word slice (1 : - 1) → "otato" (str)

return f palindrome (v middle) → True

return False

First and last letters match

Code example

```
Called 4 times ━━━━ < 1 >
function v palindrome ( v word )
    v word → "rotator" (str)
    if f len ( v word ) < 2 → False
        return True
    if v word item ( 0 ) == v word item ( -1 ) → True
        assign v middle = v word slice ( 1 : - 1 ) → "otato" (str)
        return f palindrome ( v middle ) → True
    return False
```

Middle bit is
"otato"

Code example

```
Called 4 times ━━━━ < 1 >
function v palindrome ( v word )
    v word → "rotator" (str)
    if f len ( v word ) < 2 → False
        return True
    if v word item ( 0 ) == v word item ( -1 ) → True
        assign v middle = v word slice ( 1 : -1 ) → "otato" (str)
        return f palindrome ( v middle ) → True
    return False
```

Call the
palindrome
function again!

Code example

Called 4 times

```
function v| palindrome| ( v| word| )  
    v| word| → "otato" (str)  
  
    if f| len| ( v| word| ) < 2 → False  
        return True  
  
    if v| word| item| ( 0 ) == v| word| item| ( -1 ) → True  
        assign v| middle| = v| word| slice| ( 1 : -1 ) → "tat" (str)  
        return f| palindrome| ( v| middle| ) → True  
  
    return False
```

Second time around!

word is "otato"



Code example

```
Called 4 times < 2 >
function v| palindrome| ( v| word| )
    v| word| → "otato" (str)
    if f| len| ( v| word| ) < 2 → False ←
        return True
    if v| word| item| ( 0 ) == v| word| item| ( -1 ) ← True
        assign v| middle| = v| word| slice| ( 1 : -1 ) → "tat" (str)
        return f| palindrome| ( v| middle| ) → True
    return False
```

Length not less than 2

First and last letters match again



Code example

Called 4 times

2

```
function v| palindrome( v| word )
```

```
    v| word → "otato" (str)
```

```
    if f| len( v| word ) < 2 → False
```

```
        return True
```

```
    if v| word item( 0 ) == v| word item( -1 ) → True
```

```
        assign v| middle = v| word slice( 1 : -1 ) → "tat" (str)
```

```
        return f| palindrome( v| middle ) → True
```

```
    return False
```

Middle of the word is "tat"

Code example

Called 4 times

```
function v| palindrome| ( v| word| )
```

```
    v| word| → "otato" (str)
```

```
    if f| len| ( v| word| ) < 2 → False
```

```
        return True
```

```
    if v| word| item| ( 0 ) == v| word| item| ( -1 ) → True
```

```
        assign v| middle| = v| word| slice| ( 1 : -1 ) → "tat" (str)
```

```
        return f| palindrome| ( v| middle| ) → True
```

```
    return False
```

Middle of the word is "tat"

Call palindrome function again with the word "tat"

Code example

Called 4 times

```
function v| palindrome| ( v| word| )
```

```
    v| word| → "otato" (str)
```

```
    if f| len| ( v| word| ) < 2 → False
```

```
        return True
```

```
    if v| word| item| ( 0 ) == v| word| item| ( -1 ) → True
```

```
        assign v| middle| = v| word| slice| ( 1 : -1 ) → "tat" (str)
```

```
        return f| palindrome| ( v| middle| ) → True
```

```
    return False
```

Middle of the word is "tat"

Call palindrome function again with the word "tat"

Code example

```
Called 4 times < 3 >
function v| palindrome ( v| word )
    v| word → "tat" (str)
    if f| len ( v| word ) < 2 → False
        return True
    if v| word item ( 0 ) == v| word item ( -1 ) → True
        assign v| middle = v| word slice ( 1 : -1 )
        → "a" (str)
        return f| palindrome ( v| middle ) → True
    return False
```

Now the word is "tat" it still has more than 2 letters

The middle of the word is just "a"

Call the palindrome function again!



Code example

```
Called 4 times < 4 >  
function v| palindrome ( v| word )  
    v| word → "a" (str)  
    if f| len ( v| word ) < 2 → True  
        return True → True  
    if v| word item ( 0 ) == v| word item ( -1 )  
        assign v| middle = v| word slice ( 1 : -1 )  
        return f| palindrome ( v| middle )  
    return False
```

Now the word is
"a"

This time it
has less than 2
letters!!!

Return True!

This must be a
palindrome!



Code example

```
Called 4 times ━━━━ < 3 >
function v| palindrome ( v| word )
    v| word → "tat" (str)
    if ( f| len ( v| word ) < 2 → False
        return True
    if ( v| word item ( 0 ) == v| word item ( -1 ) → True
        assign v| middle = v| word slice ( 1 : -1 ) → "a" (str)
        return f| palindrome ( v| middle ) → True
    return False
```

Which means it returns True for "tat"



Code example

```
Called 4 times < 2 >
function v palindrome( v word )
    v word → "otato" (str)
    if f len( v word ) < 2 → False
        return True
    if v word item( 0 ) == v word item( -1 ) → True
        assign v middle = v word slice( 1 : -1 ) → "tat" (str)
        return f palindrome( v middle ) → True
    return False
```

Which means it returns True for "otato"



Code example

```
Called 4 times ━━━━ < 1 >
function v palindrome ( v word )
    v word → "rotator" (str)
    if f len ( v word ) < 2 → False
        return True
    if v word item ( 0 ) == v word item ( -1 ) → True
        assign v middle = v word slice ( 1 : -1 ) → "otato" (str)
        return f palindrome ( v middle ) → True
    return False
```

AND that means it returns True for "rotator" too!

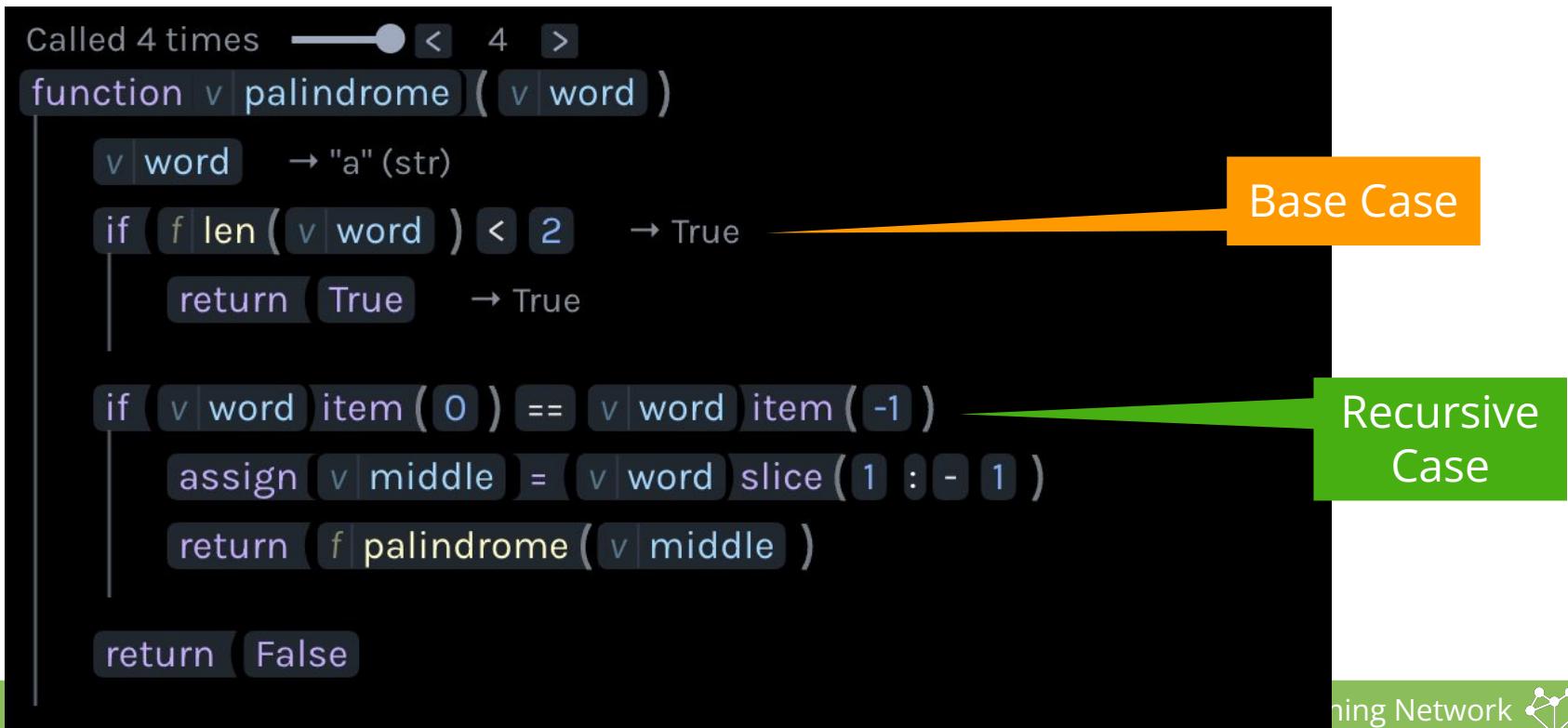
After all those layers,
we now know that "rotator" is a palindrome!

```
f | palindrome('rotator') → True
```

Base case

So if recursion keeps solving smaller and smaller versions of a problem, when does it stop?

We use a **base case** to tell the function when to stop recursing.



Project Time!

Let's get on to the next thing

Try to do the next Part!

The tutors will be around to help!