Welcome to the Labs!

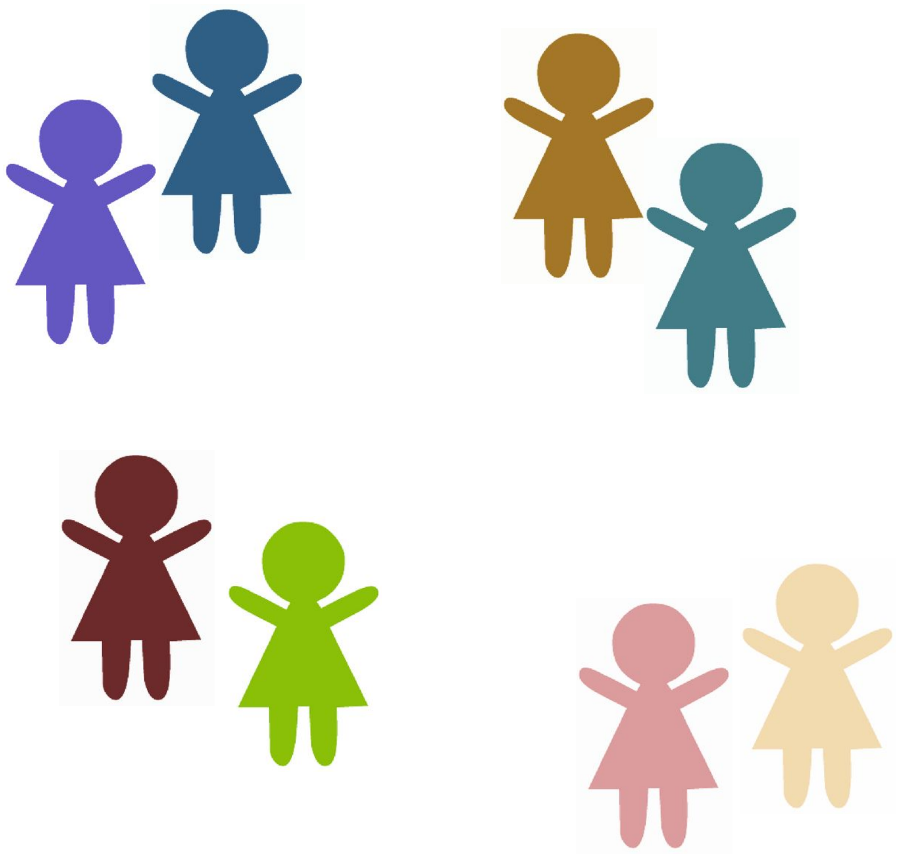
Tic Tac Toe

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!

Today's project!

Workshop Name Here

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!

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**

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 6.1: 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.

☑ **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

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.

★ **BONUS 4.3: Do some extra!**

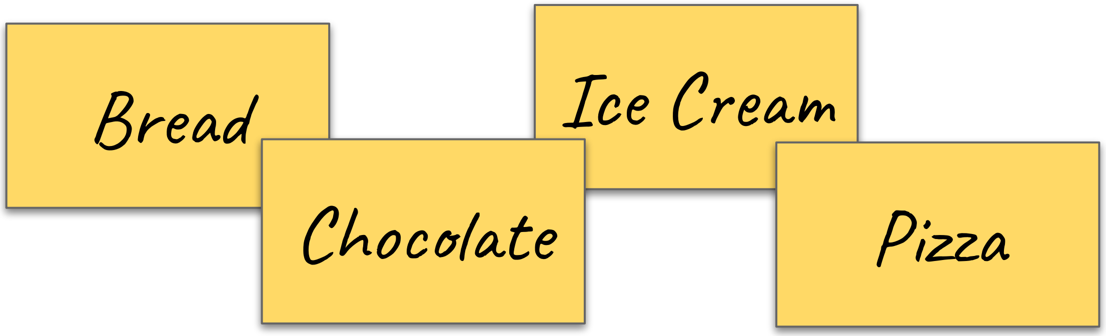
Something to try if you have spare time before the next lecture!

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!

- Bread
- Chocolate
- Ice Cream
- Pizza

Lists

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

```
>>> shopping_item1 = "Bread"
>>> shopping_item2 = "Chocolate"
>>> shopping_item3 = "Ice Cream"
>>> shopping_item4 = "Pizza"
```

So much repetition!

Instead we use a python list!

```
>>> shopping_list = ["Bread", "Chocolate", "Ice Cream",
"Pizza"]
```

You can put (almost) anything into a list

- You can have a list of integers

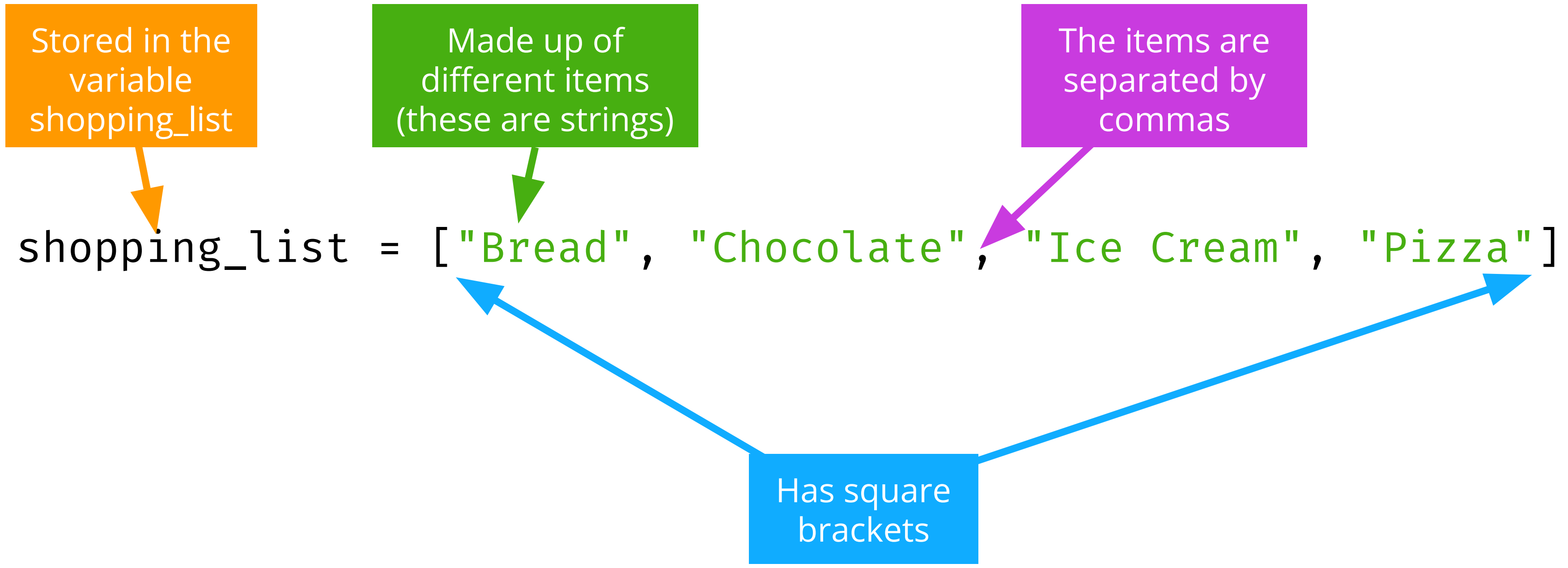
```
>>> primes = [1, 2, 3, 5, 11]
```

- You can have lists with mixed integers and strings

```
>>> mixture = [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


Stored in the variable shopping_list
Made up of different items (these are strings)
The items are separated by commas
Has square brackets

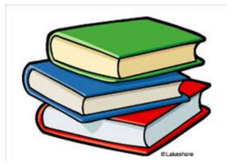
```
shopping_list = ["Bread", "Chocolate", "Ice Cream", "Pizza"]
```

Accessing Lists!

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

```
faves = ['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]`:

```
>>> faves[0]
'books'
```

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]`:

```
>>> faves[0]
'books'
```

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

```
>>> faves[1]
'butterfly'
```

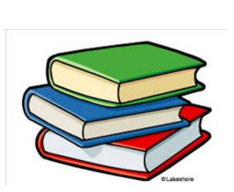
0 **[1]** 2 3

Going Negative

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

```
>>> faves[-1]
'skateboard'
```

What would `faves[-2]` return?



Going Negative

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

```
>>> faves[-1]
'skateboard'
```

What would `faves[-2]` return?

```
>>> faves[-2]
'chocolate'
```

Falling off the edge

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

```
>>> faves = ['books', 'butterfly', 'chocolate',
             'skateboard']
>>> faves[4]


Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Updating items!

We can also update things in a list:

```
>>> faves = ['books', 'butterfly',
                 'chocolate', 'skateboard']
>>> faves[2]
'chocolate'
>>> faves[2] = 'lollipops'
>>> faves
```

Updating items!

We can also update things in a list:

```
>>> faves = ['books', 'butterfly',
               'chocolate', 'skateboard']
>>> faves[2]
'chocolate'
>>> faves[2] = 'lollipops'
>>> faves
['books', 'butterfly', 'lollipops', 'skateboard']
```

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!

```
tennis_pairs = [
    ["Alex", "Emily"], ["Kass", "Annie"], ["Amara", "Viv"]
]
```

Get the first pair in the list

```
>>> first_pair = tennis_pairs[0]
>>> ["Alex", "Emily"]
```

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

```
>>> fist_player = first_pair[0]
>>> "Alex"
```

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!

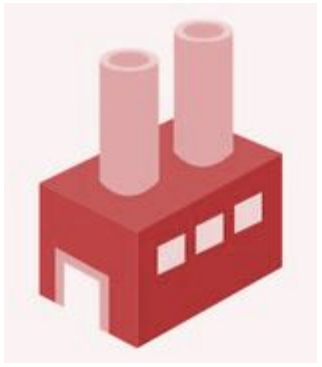
Functions!

Simpler, less repetition, easier to read code!

How functions fit together!

Functions are like factories!

Metal Worker



Your main factory!

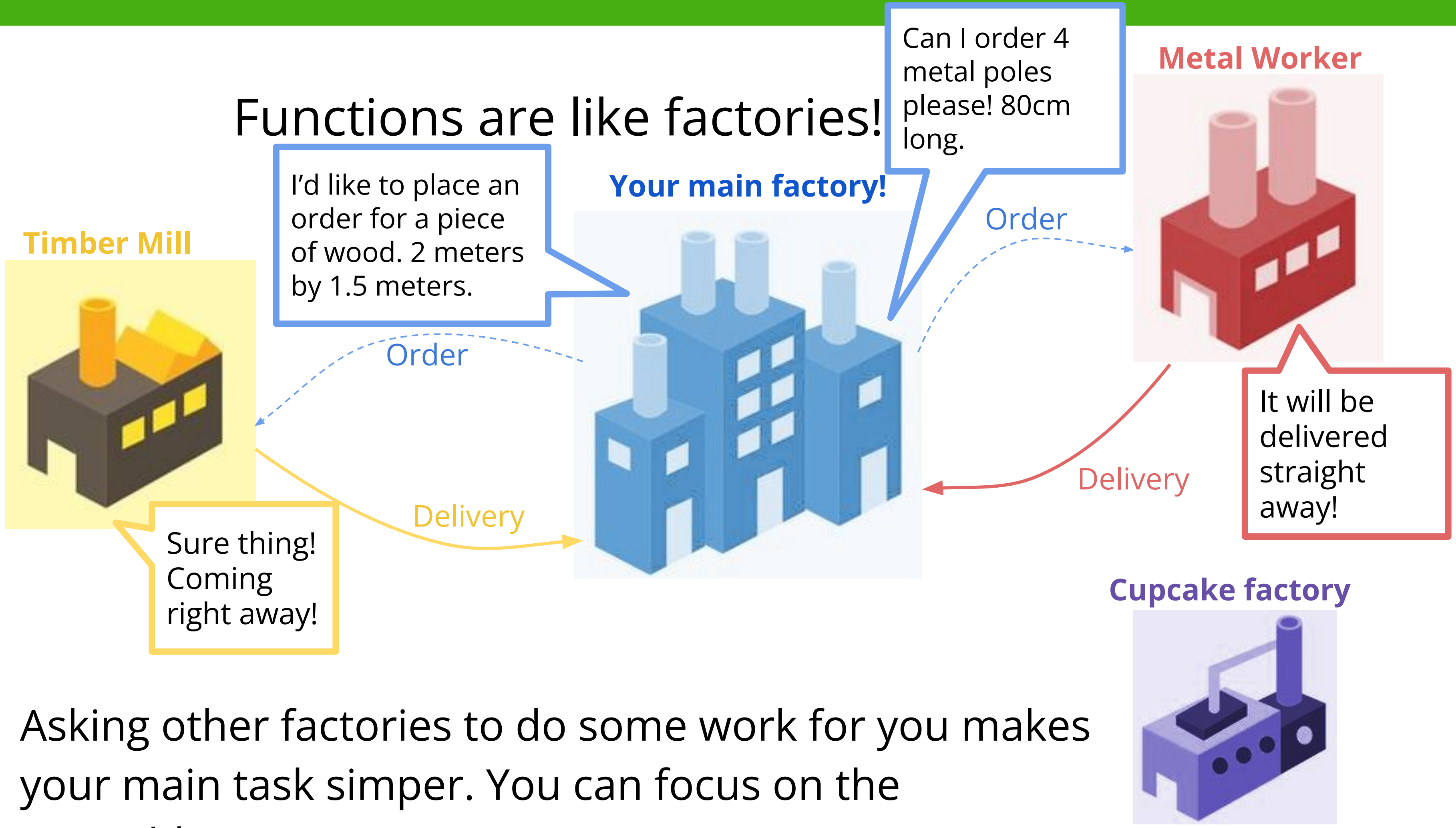
Timber Mill

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!

Functions are like factories!


Timber Mill
I'd like to place an order for a piece of wood. 2 meters by 1.5 meters.
Your main factory!
Can I order 4 metal poles please! 80cm long.
Metal Worker
Order
Order
Delivery
Delivery
It will be delivered straight away!
Sure thing! Coming right away!
Cupcake factory

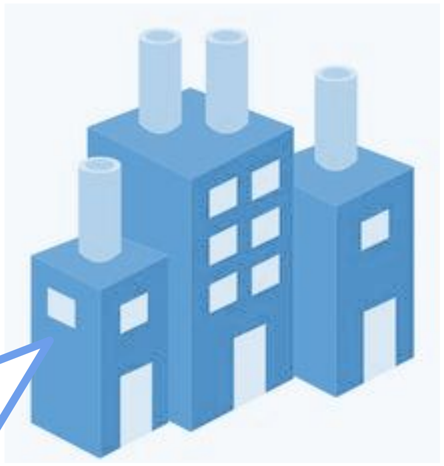
Asking other factories to do some work for you makes your main task simper. You can focus on the assembly!

How functions fit together!

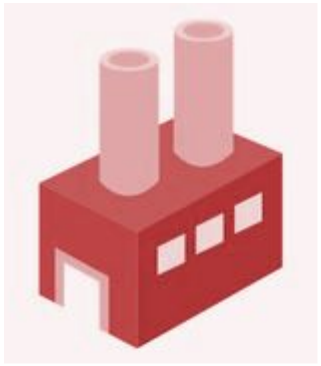
Functions are like factories!

Timber Mill

Your main factory!



Metal Worker



Cupcake factory

Look at this beautiful table I made!

Outsourcing made it simple!

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!

Helps with printing nicely

Uses stats to make decisions

Does calculations

Don’t reinvent the wheel

We’re already familiar with some python in built 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!

```
>>> len("Hello world")
11
```

Try these:

```
>>> name = "Renee"
>>> len(name)
5

>>> int("6")
6

>>> str(6)
"6"
```

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

Then you can use your function by calling it!

```
def cat_print():
    print("""
                      #
                       #
                       #
                  ^..^ #####
                  =TT=      ;
                   #########
                   # #   # #
                   M M   M M """)


cat_print()
cat_print()
```

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!

```
def cat_print():
    print("""
                         #
                          #
                          #
                ^..^ #####
                =TT=      ;
                 #########
                 # #   # #
                 M M   M M""")



cat_print()
cat_print()
```

When using a function in a **script** make sure you define the function first.

It doesn't matter if you call it from inside another function though!

Which will do this!

```
         #
          #
          #
^..^ #####
=TT=      ;
 #########
 # #   # #
 M M   M M
         #
          #
          #
^..^ #####
=TT=      ;
 #########
 # #   # #
 M M   M M
```

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)

```
>>> def hello(person):
...     print('Hello, ' + person + ', how are you?')
>>> hello('Alex')
Hello, Alex, how are you?
```

Here, we give the hello() function a name

Any string will work

```
>>> hello('abcd')
Hello, abcd, how are you?
```

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.

```
>>> def add(x, y):
...     print(x + y)
>>> add(3, 4)
7
```

Arguments stay inside the function

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

```
>>> def hello(person):
...   print('Hello, ' + person + '!')
>>> print(person)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'person' is not defined
```

Variables stay inside the function

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

```
>>> def add(x, y):
...     z = x + y
...     print(z)
>>> add(3, 4)
7
>>> z
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'z' is not defined
```

Global variables are not affected

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

```
>>> z = 1
>>> def add(x, y):
...     z = x + y
...     print(z)
>>> add(3, 4)
7
```

Global variables are not affected

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

```
>>> z = 1
>>> def add(x, y):
...     z = x + y
...     print(z)
>>> add(3, 4)
7
```

What's the value of z now?

```
>>> print(z)
```

Global variables are not affected

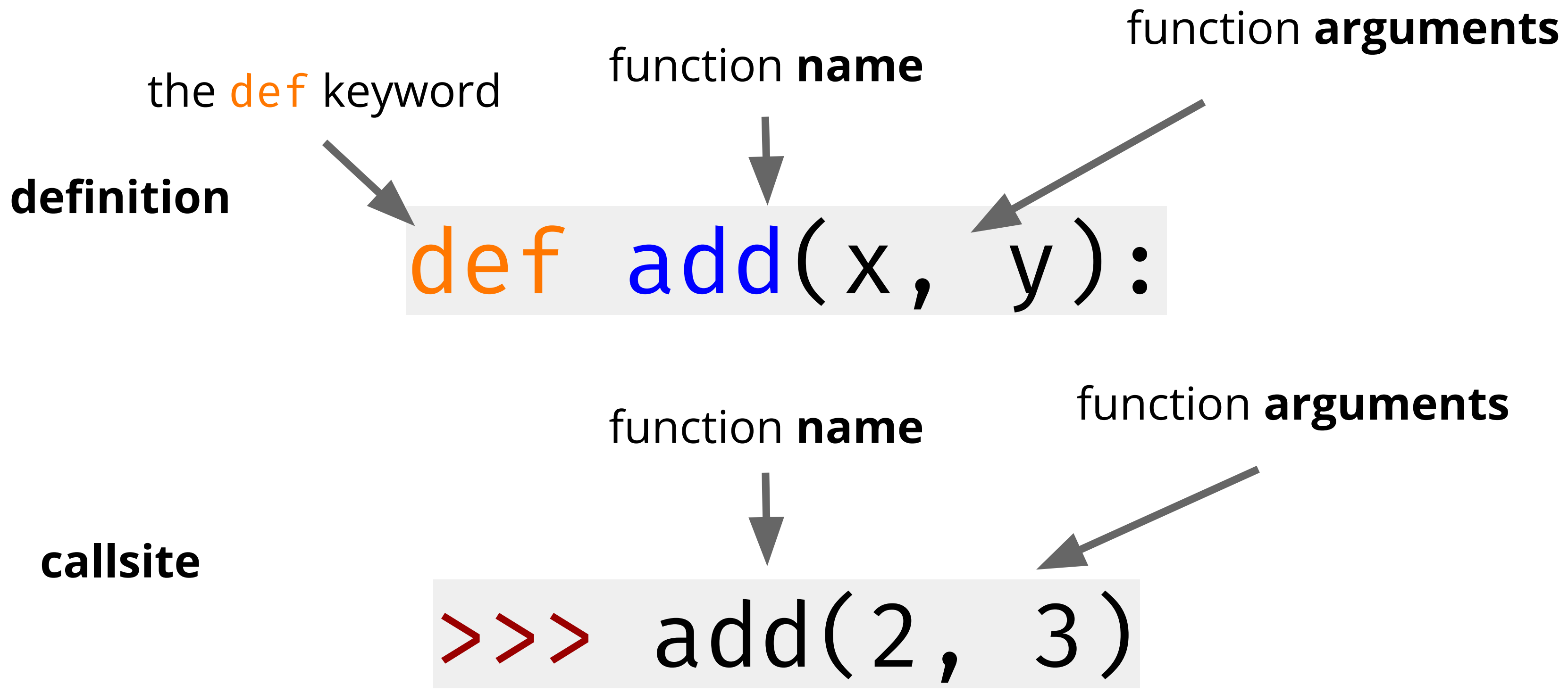
Changing a variable in a function **only changes it *inside* the function.**

```
>>> z = 1
>>> def add(x, y):
...     z = x + y
...     print(z)
>>> add(3, 4)
7
```

What's the value of z now?

```
>>> print(z)
1
```

Recap: A function signature



```
def add(x, y):
```

```
>>> add(2, 3)
```

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()

```
>>> def add(x, y):
...   print(x + y)
>>> sum = add(1, 3)
4
>>> sum
```

sum has no value!

Giving something back

Using `return` in a function immediately returns a result.

```
>>> def add(x, y):
...     z = x + y
...     return z
...
>>> sum = add(1, 3)
>>> sum
4
```

Giving something back

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

```
>>> def add(x, y):
...     print('before the return')
...     z = x + y
...     return z
...     print('after the return')
>>> sum = add(1, 3)
before the return
>>> sum
4
```

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`!

```
fave_num = 5
if fave_num < 10:
    print("that's a small number")
```

Else statements

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

```
word = "Chocolate"
if word == "GPN":
  print("GPN is awesome!")
else:
  print("The word isn’t GPN :(")
```

What happens?

Else statements

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

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

What happens?

```
>>> The word isn't GPN :(
```

Elif statements

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

```
word = "Chocolate"
if word == "GPN":
  print("GPN is awesome!")
elif word == "Chocolate":
  print("YUMMM Chocolate!")
else:
  print("The word isn’t GPN :(")
```

What happens?

Elif statements

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

```
word = "Chocolate"
if word == "GPN":
  print("GPN is awesome!")
elif word == "Chocolate":
  print("YUMMM Chocolate!")
else:
  print("The word isn’t GPN :(")
```

What happens?

```
>>> YUMMM Chocolate!
```

Booleans (True and False)

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

```
>>> "A" in "AEIOU"
>>> "Z" in "AEIOU"
>>> "a" in "AEIOU"
```

```
>>> animals = ["cat", "dog", "goat"]
>>> "banana" in animals
>>> "cat" in animals
```

```
>>> phone_book = {"Maddie": 111, "Lucy": 222, "Julia": 333}
>>> "Maddie" in phone_book
>>> "Gabe" in phone_book
>>> 333 in phone_book
```

Booleans (True and False)

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

```
True   "A" in "AEIOU"
False  "Z" in "AEIOU"
False  "a" in "AEIOU"
```

```
>>> animals = ["cat", "dog", "goat"]
False  "banana" in animals
True   "cat" in animals
```

```
>>> phone_book = {"Maddie": 111, "Lucy": 222, "Julia": 333}
True   "Maddie" in phone_book
False  "Gabe" in phone_book
False  333 in phone_book
```

It only checks in the keys!

Project Time!

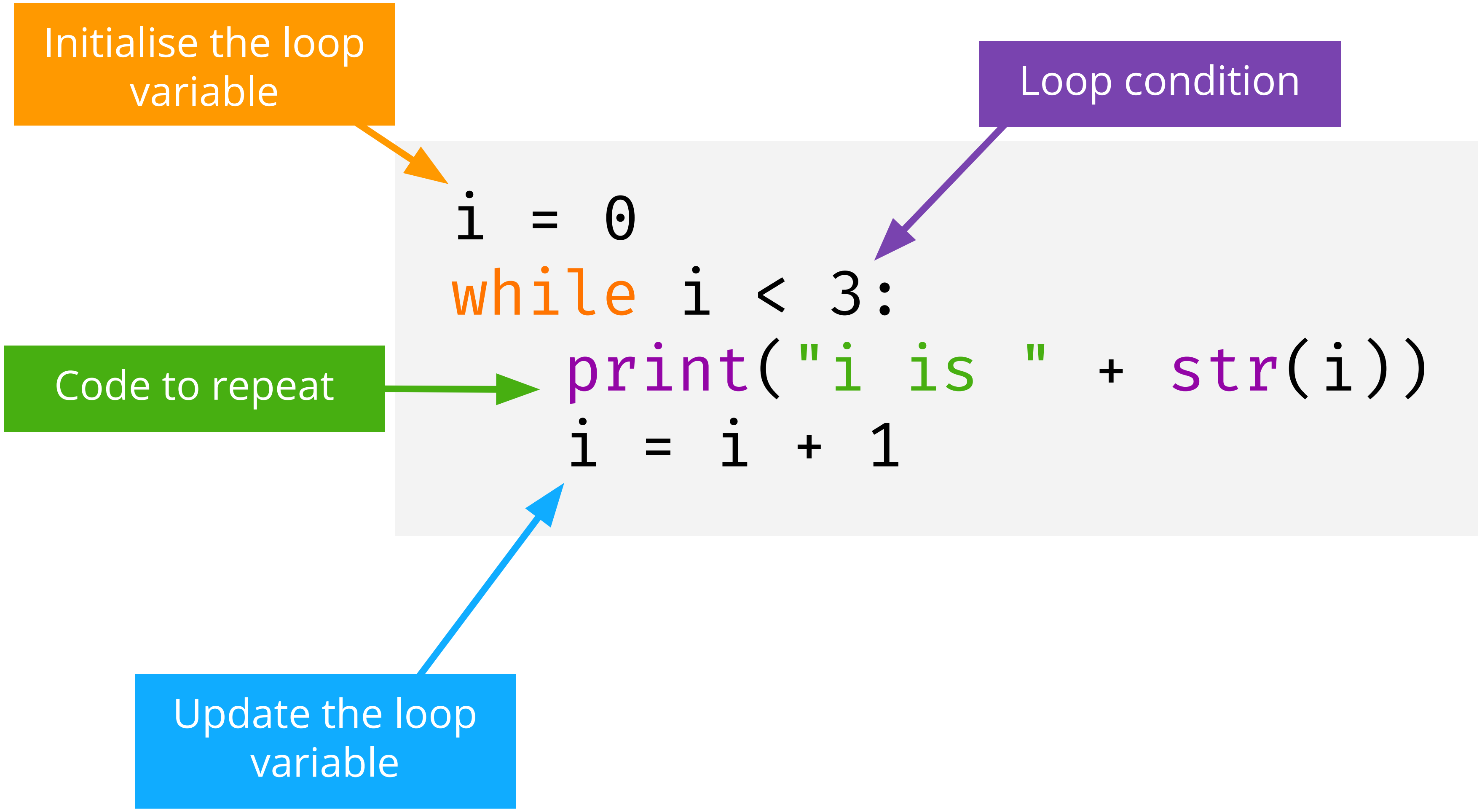
You now know all about `if`!

See `if` you can do the next Part

The tutors will be around to help!

While Loops

Introducing ... `while` loops!



```
i = 0
while i < 3:
    print("i is " + str(i))
    i = i + 1
```

What happens when.....

What happens if we forget to update the loop variable?

```
i = 0
while i < 3:
    print("i is " + str(i))
```

What happens when.....

What happens if we forget to update the loop variable?

```
i = 0
while i < 3:
    print("i is " + str(i))
```

```
i is 0
i is 0
i is 0
i is 0
i is 0
i is 0
i is 0
i is 0
i is 0
i is 0
i is 0
i is 0
i is 0
```

Give me a break!

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

That's when we use the break keyword!

```
number = 0
while number != 42 :
    number = input("Guess a number: ")

    if number = "I give up":
        print("The number was 42")
        break

    number = int(number)
```

Continuing on

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

```
number = 0
while number != 42 :
    number = input("Guess a number:  ")

    if not number.isnumeric():
        print("That’s not a number!")
        print("Try again")
        continue

    number = int(number)
```

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?

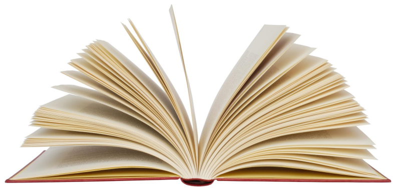
```
words = ['This', 'is', 'a', 'sentence']


print(words[0])
print(words[1])
print(words[2])
print(words[3])
```

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

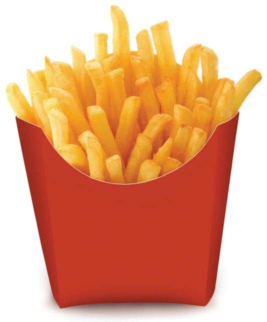
For Loops

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:

```
numbers = [1, 2, 3, 4]
for i in numbers:
    print(i)
```

What's going to happen?

Looping over a list of ints

We can loop through a list:

```
numbers = [1, 2, 3, 4]
for i in numbers:
    print(i)
```

What's going to happen?

```
>>> 1
>>> 2
>>> 3
>>> 4
```

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

Project Time!

Now you know how to use a for loop!

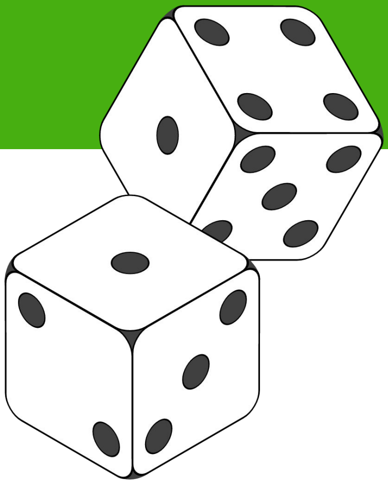
Try to do Part 5

...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!

We want the computer to be random sometimes!

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

Using the random module

Let's choose something randomly from a list!

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

Try this!

1. Import the random module!

```
>>> import random
```

2. Copy the shopping list into IDLE

```
>>> shopping_list = ["Bread", "Chocolate", "Ice Cream",
     "Pizza"]
```

3. Choose randomly! Try it a few times!

```
>>> random.choice(shopping_list)
```

Using the random module

You can also assign your random choice to a variable

```
>>> import random
>>> shopping_list = ["Bread", "Chocolate", "Ice Cream",
     "Pizza"]
>>> random_food = random.choice(shopping_list)
>>> print(random_food)
```

Project Time!

Raaaaaaaaaandom! 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!

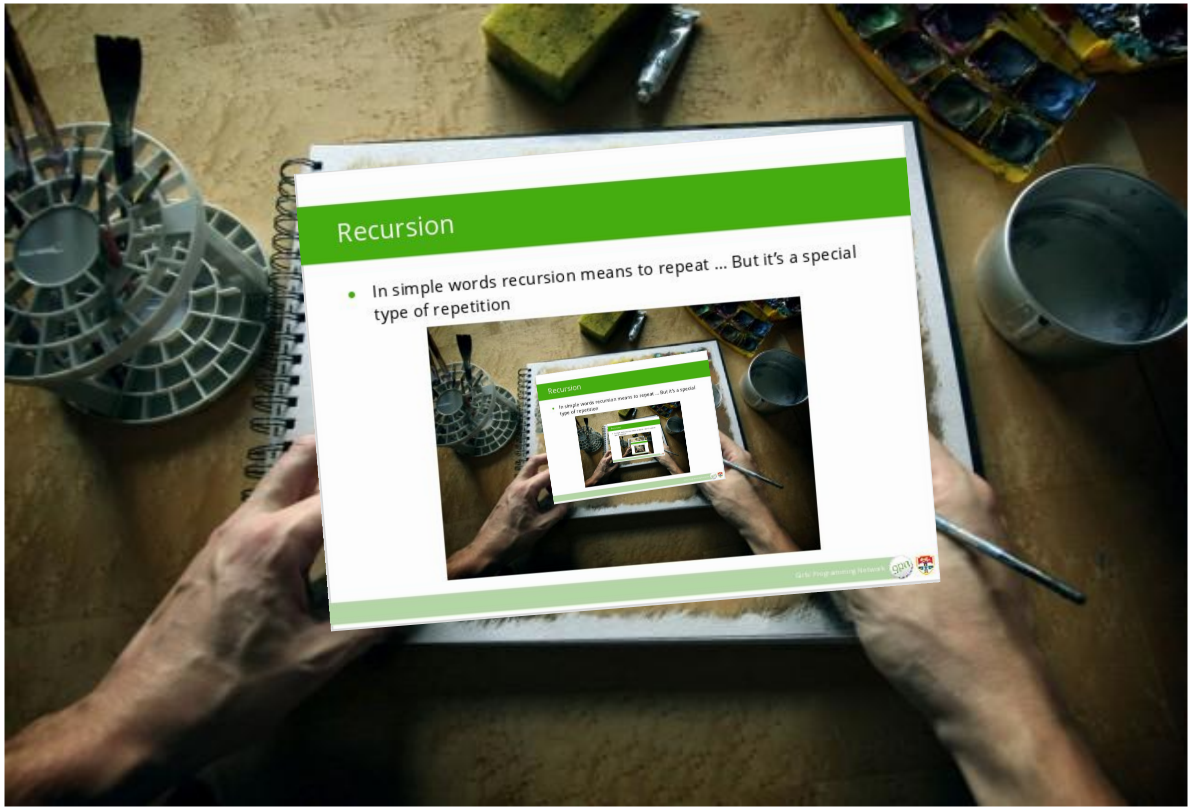
Recursion

Outline

1. What is recursion
2. Recursive function

Recursion

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

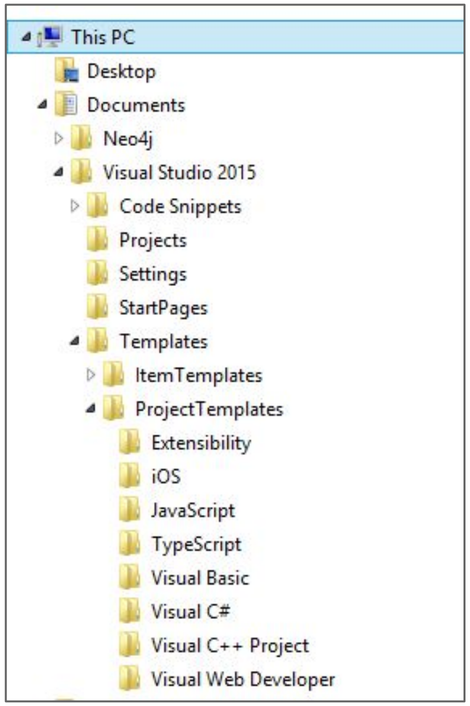

Recursion

- 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.

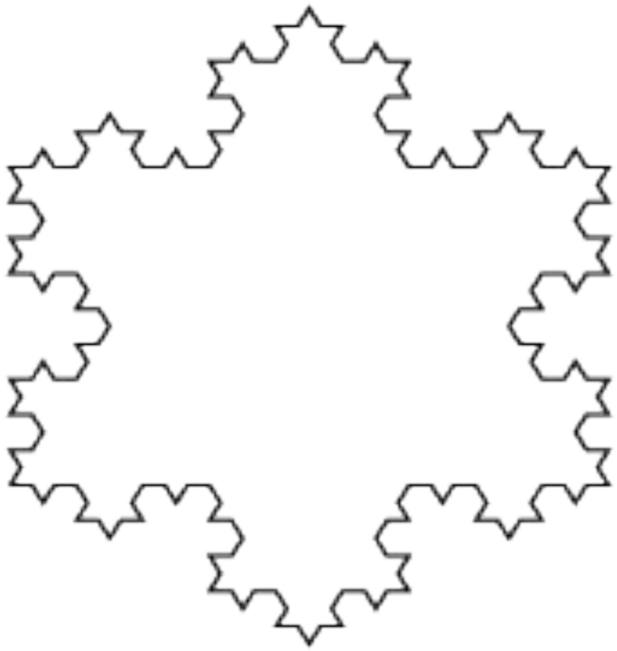
“A folder is a structure that holds files and (smaller) folders”



```
This PC
  Desktop
  Documents
    Neo4j
    Visual Studio 2015
      Code Snippets
      Projects
      Settings
      StartPages
      Templates
        ItemTemplates
        ProjectTemplates
          Extensibility
          iOS
          JavaScript
          TypeScript
          Visual Basic
          Visual C#
          Visual C++ Project
          Visual Web Developer
```

Drawing a snowflake

- How could we draw a snowflake using recursion?



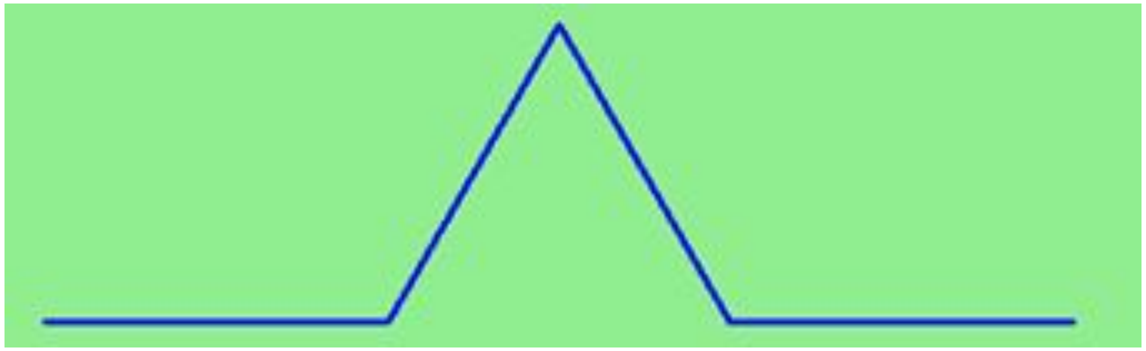
Drawing a snowflake

1

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

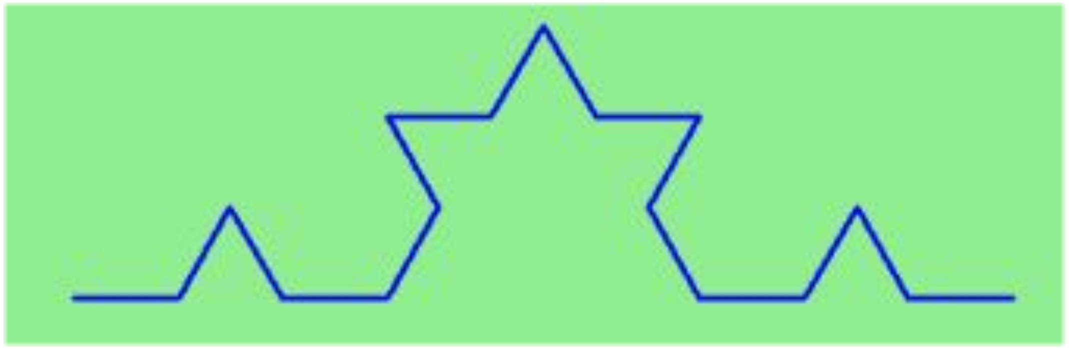
Drawing a snowflake

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



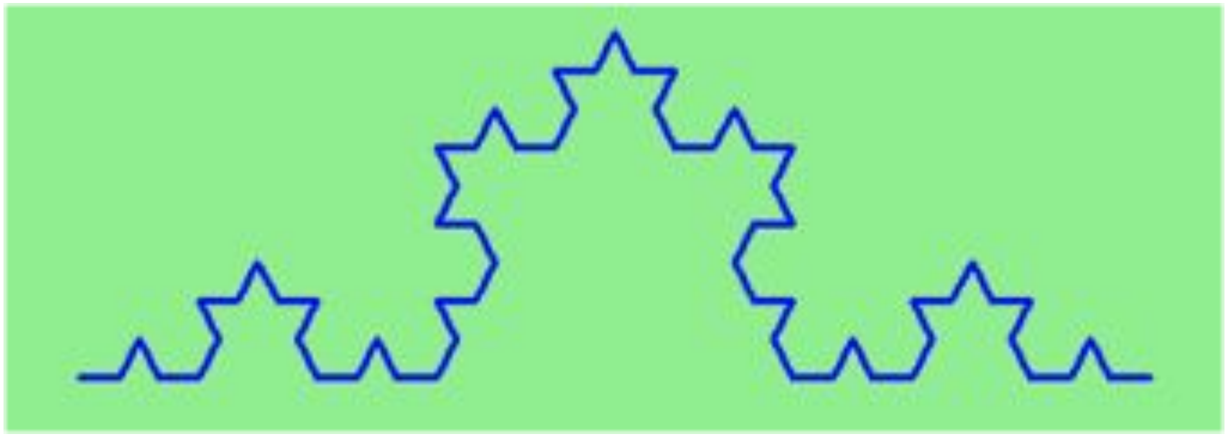
Drawing a snowflake

- 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

$\boxed{\texttt{anna}}$ `banana` $\boxed{\texttt{kayak}}$ $\boxed{\texttt{rotator}}$ `water gpn`

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

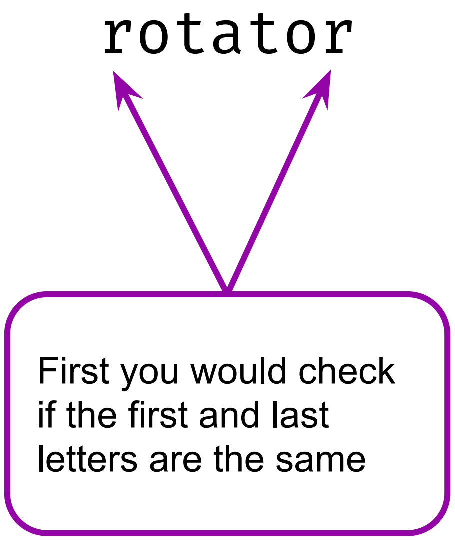
Code example

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

```
rotator
```

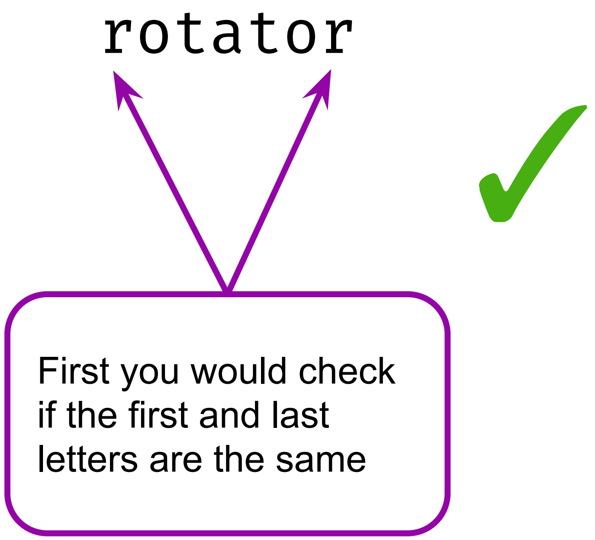
Code example

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:



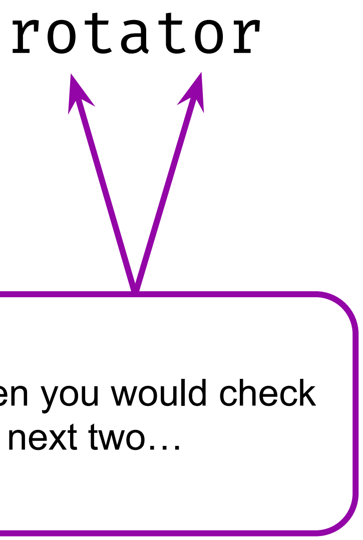
```
rotator
```

✓

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

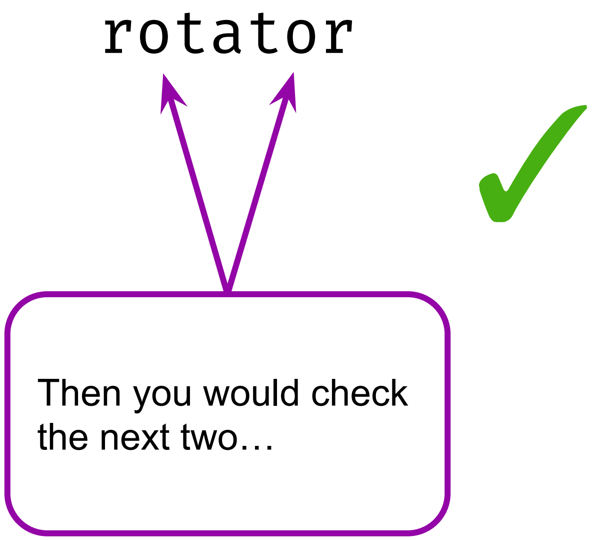
Code example

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
```

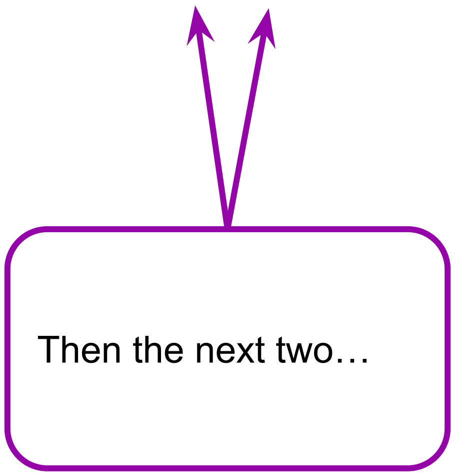
✓

Then you would check the next two…

Code example

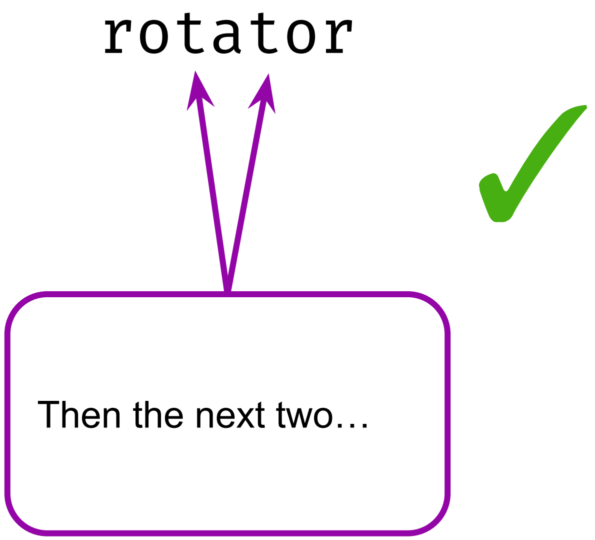
How would you work this out by hand? Let’s use this word as an example:

`rotator`



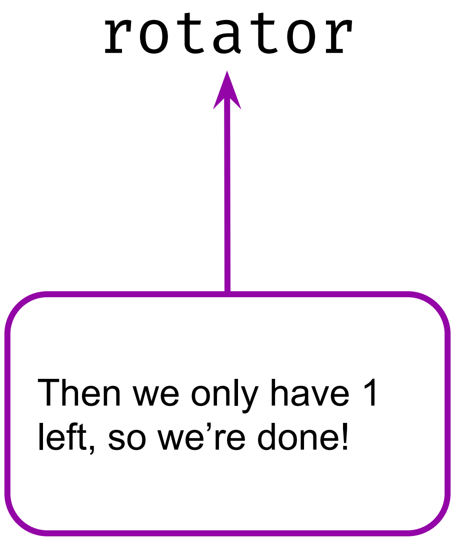
Code example

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.

```
def is_a_palindrome(word):
    if len(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 “a” is a palindrome (and this function knows it!)

```
def is_a_palindrome(word):
    if len(word) < 2:
        return True
```

Next we need to think about the next size up for our problem.
E.g. the word “wow”

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)

```
def is_a_palindrome(word):
    if len(word) < 2:
        return True
    if word[0] == word[-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)

```
def is_a_palindrome(word):
    if len(word) < 2:
        return True
    if word[0] == word[-1]:
        return is_a_palindrome(word[1:-1])
    else:
        return False
```

We recursively call our function, making the word smaller and smaller until we reach the base case.

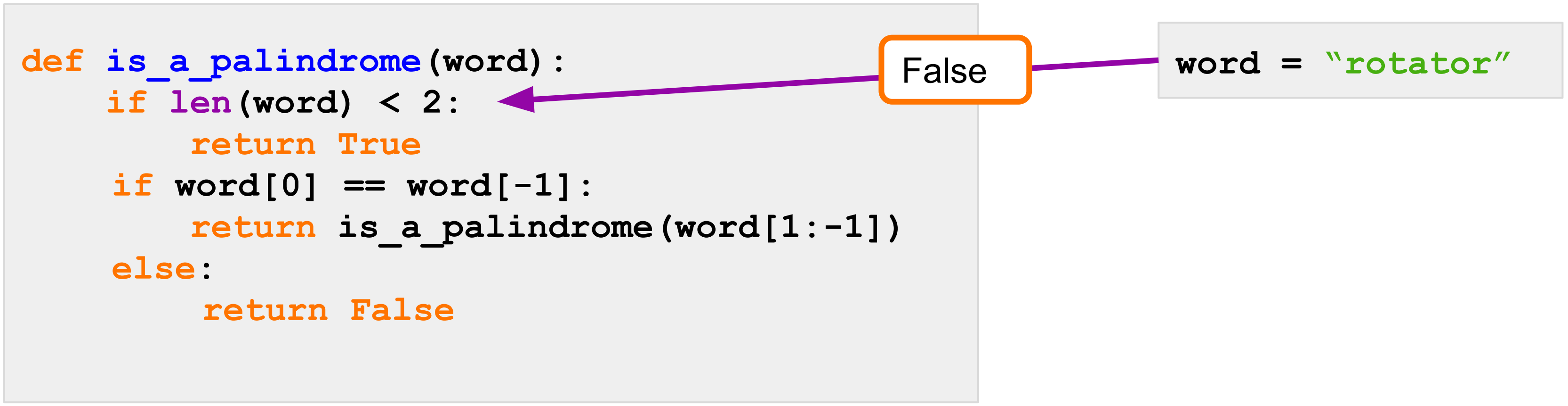
Let's look at it in more detail!

Code example

```
def is_a_palindrome(word):
    if len(word) < 2:
        return True
    if word[0] == word[-1]:
        return is_a_palindrome(word[1:-1])
    else:
        return False
```

```
word = “rotator”
```

Code example



```
def is_a_palindrome(word):
    if len(word) < 2:
        return True
    if word[0] == word[-1]:
        return is_a_palindrome(word[1:-1])
    else:
        return False
```

```
word = "rotator"
```

Code example

```
def is_a_palindrome(word):
    if len(word) < 2:
        return True
    if word[0] == word[-1]:
        return is_a_palindrome(word[1:-1])
    else:
        return False
```

```
word = "rotator"
```

Code example

```
def is_a_palindrome(word):
      if len(word) < 2:
            return True
      if word[0] == word[-1]:
            return is_a_palindrome(word[1:-1])
      else:
            return False
```

```
word = "rotator"
```

Equal?
True!

Code example

```
def is_a_palindrome(word):
    if len(word) < 2:
        return True
    if word[0] == word[-1]:
        return is_a_palindrome(word[1:-1])
    else:
        return False
```

```
word = "rotator"
```

Code example

```
def is_a_palindrome(word):
    if len(word) < 2:
        return True
    if word[0] == word[-1]:
        return is_a_palindrome(word[1:-1])
    else:
        return False
```

```
word = "otato"
```

Code example

```
def is_a_palindrome(word):
    if len(word) < 2:
        return True
    if word[0] == word[-1]:
        return is_a_palindrome(word[1:-1])
    else:
        return False
```

False

```
word = "otato"
```

Code example

```
def is_a_palindrome(word):
    if len(word) < 2:
        return True
    if word[0] == word[-1]:
        return is_a_palindrome(word[1:-1])
    else:
        return False
```

```
word = “otato”
```

Code example

```python
def is_a_palindrome(word):
    if len(word) < 2:
        return True
    if word[0] == word[-1]:
        return is_a_palindrome(word[1:-1])
    else:
        return False
```

```
word = "otato"
```

Equal?
True!

Code example

```
def is_a_palindrome(word):
    if len(word) < 2:
        return True
    if word[0] == word[-1]:
        return is_a_palindrome(word[1:-1])
    else:
        return False
```

```
word = “otato”
```

Code example

```
def is_a_palindrome(word):
    if len(word) < 2:
        return True
    if word[0] == word[-1]:
        return is_a_palindrome(word[1:-1])
    else:
        return False
```

```
word = "tat"
```

Code example

```
def is_a_palindrome(word):
    if len(word) < 2:
        return True
    if word[0] == word[-1]:
        return is_a_palindrome(word[1:-1])
    else:
        return False
```

False

```
word = "tat"
```

Code example

```
def is_a_palindrome(word):
    if len(word) < 2:
        return True
    if word[0] == word[-1]:
        return is_a_palindrome(word[1:-1])
    else:
        return False
```

```
word = "tat"
```

Code example

```
def is_a_palindrome(word):
    if len(word) < 2:
        return True
    if word[0] == word[-1]:
        return is_a_palindrome(word[1:-1])
    else:
        return False
```

```
word = "tat"
```

Equal?
True!

Code example

```
def is_a_palindrome(word):
    if len(word) < 2:
        return True
    if word[0] == word[-1]:
        return is_a_palindrome(word[1:-1])
    else:
        return False
```

```
word = “tat”
```

Code example

```
def is_a_palindrome(word):
    if len(word) < 2:
        return True
    if word[0] == word[-1]:
        return is_a_palindrome(word[1:-1])
    else:
        return False
```

```
word = "a"
```

Code example

```
def is_a_palindrome(word):
    if len(word) < 2:
        return True
    if word[0] == word[-1]:
        return is_a_palindrome(word[1:-1])
    else:
        return False
```

True

```
word = “a”
```

Code example

```
def is_a_palindrome(word):
    if len(word) < 2:
        return True
    if word[0] == word[-1]:
        return is_a_palindrome(word[1:-1])
    else:
        return False
```

True

```
word = "tat"
```

Code example

```
def is_a_palindrome(word):
    if len(word) < 2:
        return True
    if word[0] == word[-1]:
        return is_a_palindrome(word[1:-1])
    else:
        return False
```

True

```
word = "otato"
```

Code example

```
def is_a_palindrome(word):
    if len(word) < 2:
        return True
    if word[0] == word[-1]:
        return is_a_palindrome(word[1:-1])
    else:
        return False
```

True

```
word = "rotator"
```

Code example

```
def is_a_palindrome(word):
     if len(word) < 2:
          return True
     if word[0] == word[-1]:
          return is_a_palindrome(word[1:-1])
     else:
          return False
```

```
word = "rotator"
```

True

"rotator" is a palindrome!

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.

```
def is_palindrome(word) :
    if len(word) < 2:
        return True
    if word[0] == word[-1]:
            return is_palindrome(word[1:-1])
```

Base Case

Recursive Case

Project Time!

Let’s get on to the next thing

Try to do the next Part!

The tutors will be around to help!