NIHR | Applied Research Collaboration South West Peninsula

@penCHORD_UoE
@peninsula_ARC

Module 1 : Introduction to OR, Data Science and Programming
Recap & Revision Session 1 : Welcome to Python
Elliott Coyne

"Practice Makes Perfect!"

HSMA 5

#hsma5isalive

Getting to Know Each Other

- Name
- What part of the country you are currently living in
- Work background (NHS/ Police/ Social Care)
- What Peer Support Group (PSG) you are in
- What did you want to be when you grew up? (Feel free to make us laugh!)

Tech and Apps Used...

- Slack
  - Channels to read and why
- Github
  - How to download the materials

Learning Objectives

- Lets hear them!!

(Re) Introducing….

Back to Basics

1D

Hello World!

There is a tradition in Computer Science that the first program anyone should write in a Programming Language is one that writes the words “Hello World!” to the screen.

(Look, back in my day (the 80s), getting a machine to display *anything* on your TV was pretty amazing stuff!)

Let’s write a Hello World program in Python

The print() Function

The print() function displays text on the screen. The input to the function is what we want it to print.

Writing the Hello World! Program

In your script window (on the left), type the following :

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

Push F5 (or the big green Play button on the top toolbar) to run the program. Observe the output.
Exciting, yes? :)

(We could have also written the print command in the Interactive Console, and it would have run the command immediately. Give it a try!)

(Re) Introducing….

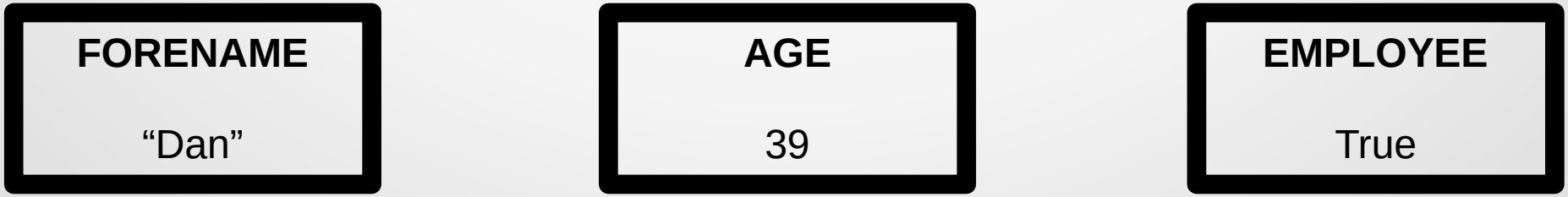
Variable Types

Variables

Often, when programming, we need to store pieces of information for later use.

We store this information in the computer's *memory*.

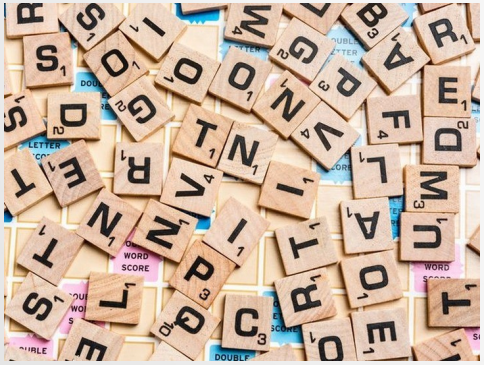
We organise this information into "boxes" called *variables.* Each variable has a *name* and a *value*.



Variable Types

Information stored in variables come in all shapes and flavours, known as *types :*

Numbers
Integers (whole numbers)
Floating Point Numbers (with decimal places)



Text
Characters (individual letters, numbers and symbols)
Strings (sequences of letters, numbers and symbols)

Boolean
Data type that indicates if something is *True* or *False*

Collections
Lists (ordered sequence of things)
Sets (unordered sequence of unique things)
Dictionaries (unordered collection of pairs of “index” and “value”)
Tuples (like a list, but can never change)

Variable declarations

In many programming languages, we need to create a variable by *declaring* it. This means specifying its type alongside its name.

In Python, variables are *dynamic*, so we don't need to do this. Instead, we simply assign a value to a name to create the variable, and the type will be automatically set based on the value. We assign values in Python using the = operator. It basically says "Let *this* have a value of *this*".
e.g.

```
age = 52
name = "Bob"
```

What types will Python assign to these variables?

Variable Types

Let's remind ourselves of the main variable types, and see how we use them in Python :

Numbers

int – integer numbers (whole numbers) `my_int = 27`

float – floating point numbers (numbers with up to 15 decimal places) `my_float = 32.451`

Strings

Sequences of characters. Denoted using " or '.

```
my_str_1 = "Apple"

my_str_2 = 'Dan is the best teacher'
```

List

Ordered sequences of items denoted using [ ].

```
my_list = [1, 5, 3, "Matthew", True]
```

Variable Types

Set

Unordered sequences of *unique* items denoted using { }.

```
my_set = {4, 1, 3, 5, 2}
```

Dictionary

Unordered collection of key-value pairs denoted using {x:y}

```
my_dict = {"Name":"Dan", "Age":39}
```

Tuple

Ordered sequences of items, like a list, but immutable (once created, cannot be changed). Denoted using ( )

```
my_tuple = ("Test", 1, 7, True, "Orange")
```

1C

Boolean

Boolean values are either *True* or *False*. `my_boolean = False`

Dictionaries

A dictionary is a really useful way to store values associated with an index value. We may want to store various information about a patient in a dictionary so it becomes easier to identify to what each element refers :

```
my_patient_dictionary = {"patient_ID":1674234,
                         "name":"Bob Smith",
                         "year_of_birth":1961,
                         "readmission":True
                        }

print (my_patient_dictionary)

# We can easily grab a particular piece of information
# about our patient # using [] notation
print (my_patient_dictionary["name"])

# And we can change dictionary values easily
my_patient_dictionary["year_of_birth"] = 1960
print (my_patient_dictionary)

# Or even add new key-value pairs
my_patient_dictionary["month_of_birth"] = 6
print (my_patient_dictionary)

# Or delete one we no longer need
del my_patient_dictionary["readmission"]
print (my_patient_dictionary)
```

```
{'patient_ID': 1674234, 'name': 'Bob Smith', 'year_of_birth': 1961, 'readmission': True}
Bob Smith
{'patient_ID': 1674234, 'name': 'Bob Smith', 'year_of_birth': 1960, 'readmission': True}
{'patient_ID': 1674234, 'name': 'Bob Smith', 'year_of_birth': 1960, 'readmission': True, 'month_of_birth': 6}
{'patient_ID': 1674234, 'name': 'Bob Smith', 'year_of_birth': 1960, 'month_of_birth': 6}
```

Mathematical Operators

It's usually the case we want to manipulate our variable values in some way. When dealing with numbers, we will likely want to apply common mathematical functions to them :

```
a + b # a plus b
a - b # a minus b
a * b # a multiplied by b
a / b # a divided by b
a % b # a modulus b (return the remainder from a divided by b)
a ** b # a to the power of b
```

```
a += 1 # this is shorthand for a = a + 1
a -= 1 # this is shorthand for a = a - 1
```

Comparison Operators

There are many comparison operators in Python. They help form relational statements that return a *Boolean* value of *True* or *False.*

```
a == b # value of a is equal to value of b
a != b # value of a is not equal to value of b
a > b # value of a is greater than value of b
a < b # value of a is less than value of b
a >= b # value of a is greater than or equal to value of b
a <= b # value of a is less than or equal to value of b
a == b and a < c # value of a is equal to value of b AND less than value of c
a == b or a < c # value of a is equal to value of b OR less than value of c
```

Dealing with Spaces in Names

In programming languages, a space indicates a separation between instructions, values etc. So if we want to name something (such as a variable) with multiple words (as we do frequently) then we can't use spaces. There are two main conventions for dealing with spaces :

camelCase

snake_case

You can use whichever one you prefer. My preference is for snake_case and so that's what you'll see in most of the course.

(Re) Introducing….

Lists

1D

Working with Lists : Create and Append

Lists are very useful ways to store multiple items of data, to which we can refer later.

```
# Define a new empty list
my_empty_list = []

# Define a new list with some starting elements
my_list = [3, 4, 7, "hello"]

# To add an item to a list, we use the append function
my_list.append(2)

print (my_list)
```

```
[3, 4, 7, 'hello', 2]
```

Working with Lists : Extend

We can also use the extend function to add multiple items to a list – we give it a list, and it adds all the items in that list to the list to which we want to add :

```
my_list = [1,2,3,4,5]
my_list.extend([6,7,8,9,10])
print(my_list)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

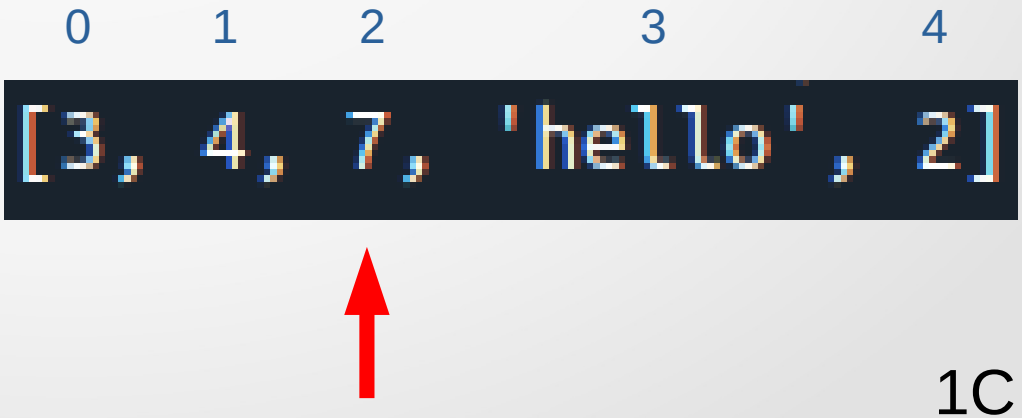
Working with Lists : Indices

To refer to a specific item in a list, we use [x] notation, where x is the index of the element we want to refer to.

REMEMBER : In Python, as with most programming languages, we start counting from 0. So the second element would have an index of 1.

```
print (my_list[2])
```

```
7
```



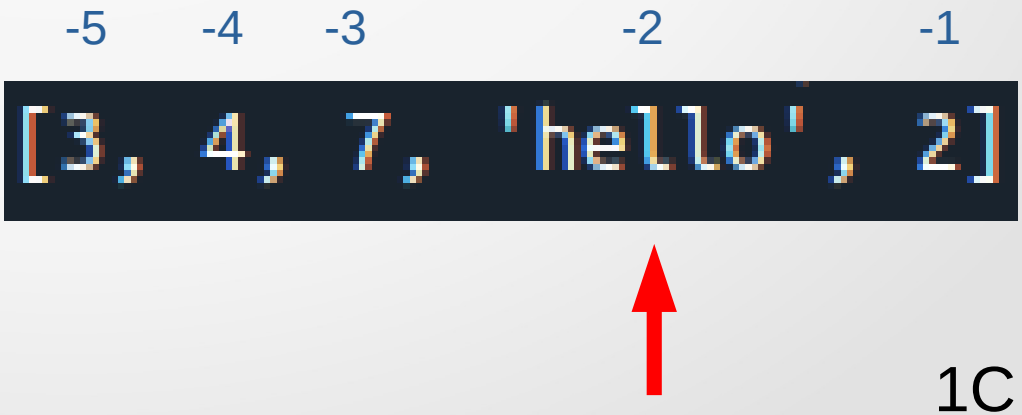
Working with Lists : Negative Indices

We can also use negative indexing to refer to an item based on its position from the end of the list.

An index of -1 would give the last item in the list.
An index of -2 would give the penultimate item in the list etc

```
print (my_list[-2])
```

```
hello
```



Working with Lists : Index Ranges

We can refer to multiple items in a list, rather than a single element.

```
my_list = [3, 4, 7, "hello", 2]

# Print everything from 2nd element up to (but not including) 4th element
print (my_list[1:3])

# Print everything up to (but not including) 3rd element
print (my_list[:2])

# Print everything from 4th element onwards
print (my_list[3:])
```

```
[4, 7]
[3, 4]
['hello', 2]
```

Working with Lists : Length and Pop

```
# We can remove items from a list using either remove() or pop()
# remove() removes a specified item
my_list.remove("hello")
print (my_list)

# Let's restore the list
my_list = [3, 4, 7, "hello", 2]

# pop() removes an item specified by its index in the list
# or removes the last item in the list if no index is specified
my_list.pop(3)
print (my_list)

# Both of the above remove "hello" from the list
```

```
[3, 4, 7, 2]
[3, 4, 7, 2]
```

We can return the length of a list using the len() function :

```
my_list = [3,9,1,2]
print (len(my_list))
```

→ 4

Working with Lists : Checking existence

We can check whether an item is in a list using the *in* command.

```
my_list = [3, 4, 7, "hello", 2]

if 2 in my_list:
    print ("Already there")
```

```
Already there
```

Working with Lists : Copying Lists

It may be tempting to copy a list by saying :

```
my_list = [3, 4, 7, "hello", 2]

copy_of_my_list = my_list
```

BUT this won't work as expected. What do you think will happen?

Working with Lists : Copying Lists

The previous example will simply make another reference to the *same list*.

To create a separate copy of the list, we need to use the *copy()* function.

```
my_list = [3, 4, 7, "hello", 2]

copy_of_my_list = my_list.copy()
```

Now we have *two lists*, and can work with them separately.

(Re) Introducing….

User Input

1D

User Input

Sometimes we need to ask the user to input something in order to continue with the program.

In Python, getting input from the user is easy.

We simply tell Python that we need an input, any message we want to display to the user, and the name of the variable in which we want to store the input.

User Input Example

```
age = input("How old are you? : ")
```

The above will store the user input. But there might be a problem with it if we're storing something like an age. Does anybody know what it might be?

Casting

By default, inputs are read in as *strings* – sequences of characters that are treated as text. But for something like age, we probably want to work with it as a number.

To do this, we need to *cast* the variable as an *integer*. We do this by specifying the type to which we want to cast, and then wrapping the value we want to translate (or “cast”) in brackets.

We could read in the input and do this separately after :

```
age = input("How old are you? : ")
age = int(age)
```

or we could just do it on the same line we take the input :

```
age = int(input("How old are you? : "))
```

Exercise 1

Using what you've learned so far, write a program that asks for the user's name, then asks for their age, before printing the message :

"Happy birthday <<name>>, you are <<age>> today"

You'll have 10 minutes, so feel free to stretch your legs if you finish early!

Example

Let’s say we want to build a simple calculator that just adds up two numbers. We would need to tell the computer to :

1. Ask the user for the first number
2. Store the first number in memory as a variable
3. Ask the user for the second number
4. Store the second number in memory as a variable
5. Perform an *addition* operation on the two variable values
6. Store the result of the addition in memory as a variable
7. Display the result variable to the user.

(Re) Introducing….

fStrings

fStrings

There’s a really nice feature in newer Python versions (3.6 onwards) called *fStrings* – “Formatted String Literals”.

These are strings where we can include formatting within the string to define where we want dynamic text. We often want to do this where we want to insert the value of a variable into a string of text.

e.g. “Her name is <<insert name here>>”

fStrings

To use fStrings, we simply put the character *f* ahead of the “ at the start of our string.

Then, we use { } to denote where we want to include a variable value, by giving the name of the variable.

For example :

```
f"The patient lives in {home_town}"
```

The Hello Dan Program

Let's now amend our Hello World program. Instead of saying hello to the world, we're going to make it more personal. We're going to say hello to you, personally!

Change your script to the following :

```
my_name = "Dan"
print (f"Hello {my_name}")
```

Now run the program.

More fString features

You can do more than just put variable names in the curly brackets in fStrings. You can put any instruction that results in something that could be interpreted as a string (e.g. numbers).

```
print (f"The answer is {2+2}")

my_float = 4.678235468594900127458

# .2f here means "a floating point with a precision of 2 decimal places"
print (f"My number rounded to 2 decimal places is {my_float:.2f}")

frequency = "Sometimes"
hsma_master = "Dan"
the_enemy = "THE LINE"

print (f"{frequency} we want to write very long fStrings that need to span",
       f"multiple lines if we don't want {hsma_master} to tell us off for",
       f"going over... ... {the_enemy}")
```

```
The answer is 4
My number rounded to 2 decimal places is 4.68
Sometimes we want to write very long strings that need to span multiple lines if we don't want Dan to tell us off for going over... ... THE LINE
```

The old method

Although the use of fStrings is the preferred method for including dynamic text in print statements in newer versions of Python, you will likely see examples of the older method being used too.

```
my_name = "Dan"
print ("Hello ", my_name)
```

```
my_name = "Dan"
my_age = 40

print ("Happy birthday ", my_name, ", you are ", my_age, " today.", sep="")
```

(Re) Introducing….

Conditional Logic

Conditional Logic

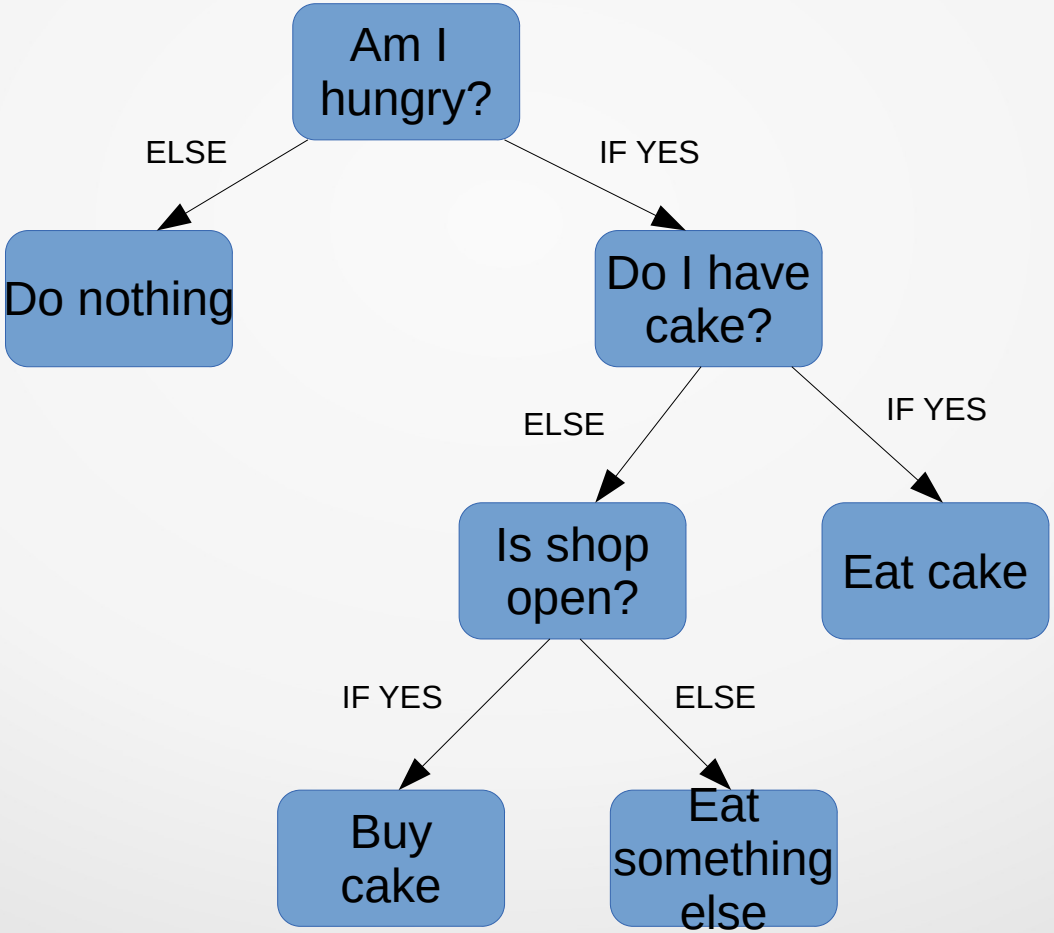
It is rare that our programs are completely linear. Often, we need to change what happens depending on whether or not something is true.

This is known as *Conditional Logic*.

Conditional Logic allows us to tell the computer “if this is true, then do this”

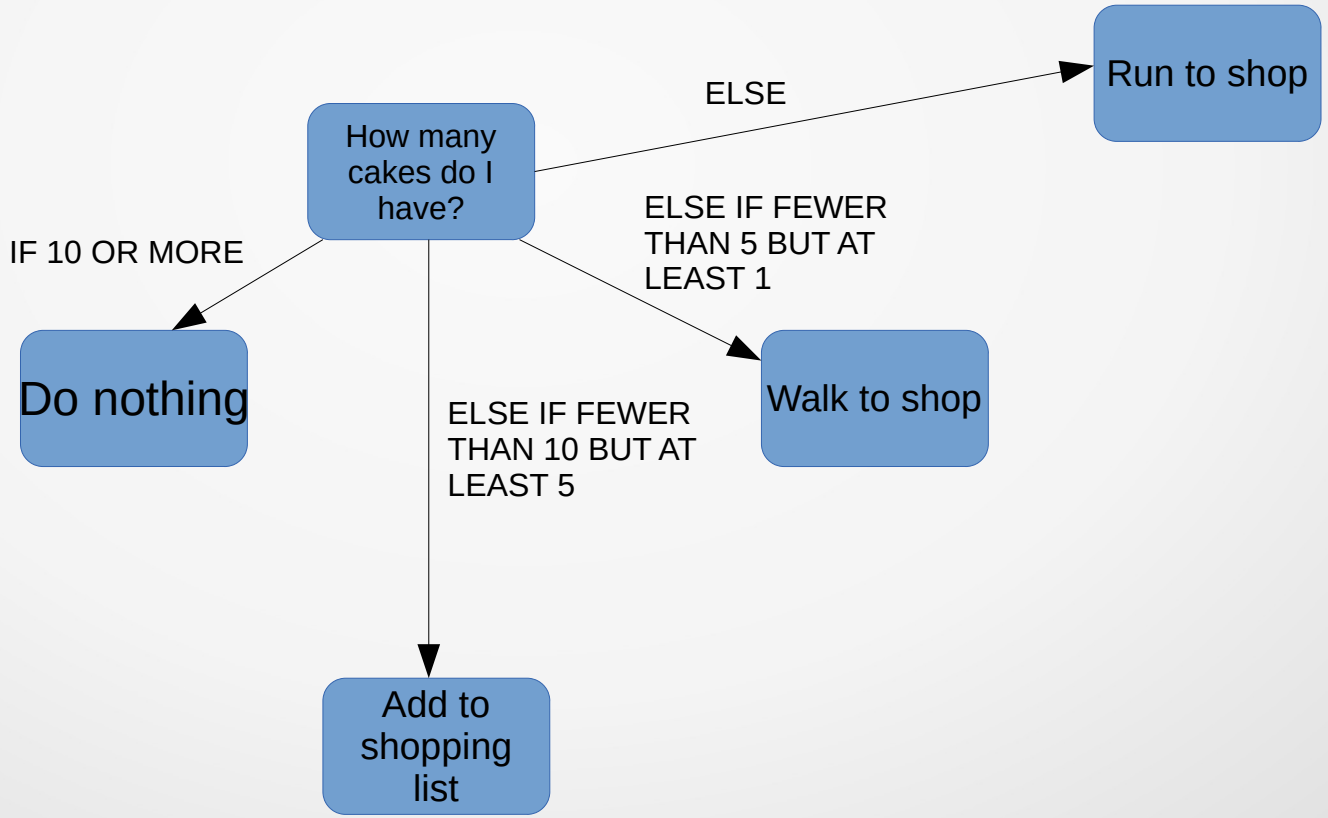
Conditional Logic

A good way to think about Conditional Logic is as a *Decision Tree* :



Conditional Logic

Sometimes we have multiple initial checks to make on the original condition :



Conditional Logic Example

Let’s imagine we need to print a different message to the user depending on whether or not someone is under 60 or aged 60+.

We might write the following :

```
if age < 60:
    print("Please go to room A")
else:
    print("Please go to room B")
```

Note the tab indentations – these are important in Python to indicate blocks of code, such as the block within a condition in an if statement

elif

Sometimes we don’t just have two conditions to check. If we have more than two, it can be useful to use the “else if” command – *elif*

```
if age < 16:
    print ("Please go to the children's room")
elif age < 60:
    print ("Please go to room A")
else:
    print ("Please go to room B")
```

Exercise 2

Let's reflect on what you've learned so far :
- Printing to the display
- Requesting and storing inputs from the user
- Variables and casting variables
- Mathematical operators
- Conditional Logic using IF, ELIF and ELSE

Switch to Jupyter Lab. Spend the next **35 minutes (+ 5 minute break)** working through the Jupyter Notebook "python_prog_workbook_1.ipynb". Remember, you can test if your code cell is working by running the cell (CTRL + Enter)

If you're unable to use Jupyter Lab, you can either :
- open Google CoLab and import the notebook to work on it in there, or
- open the file on the GitHub repository to see the questions, and write your solutions in Spyder

Work together in your groups.

(Re) Introducing….

Functions

Functions

Often, when we write things we want a computer to do, we want to do them more than once, maybe at different points in our computer program. Or even in a different program altogether.

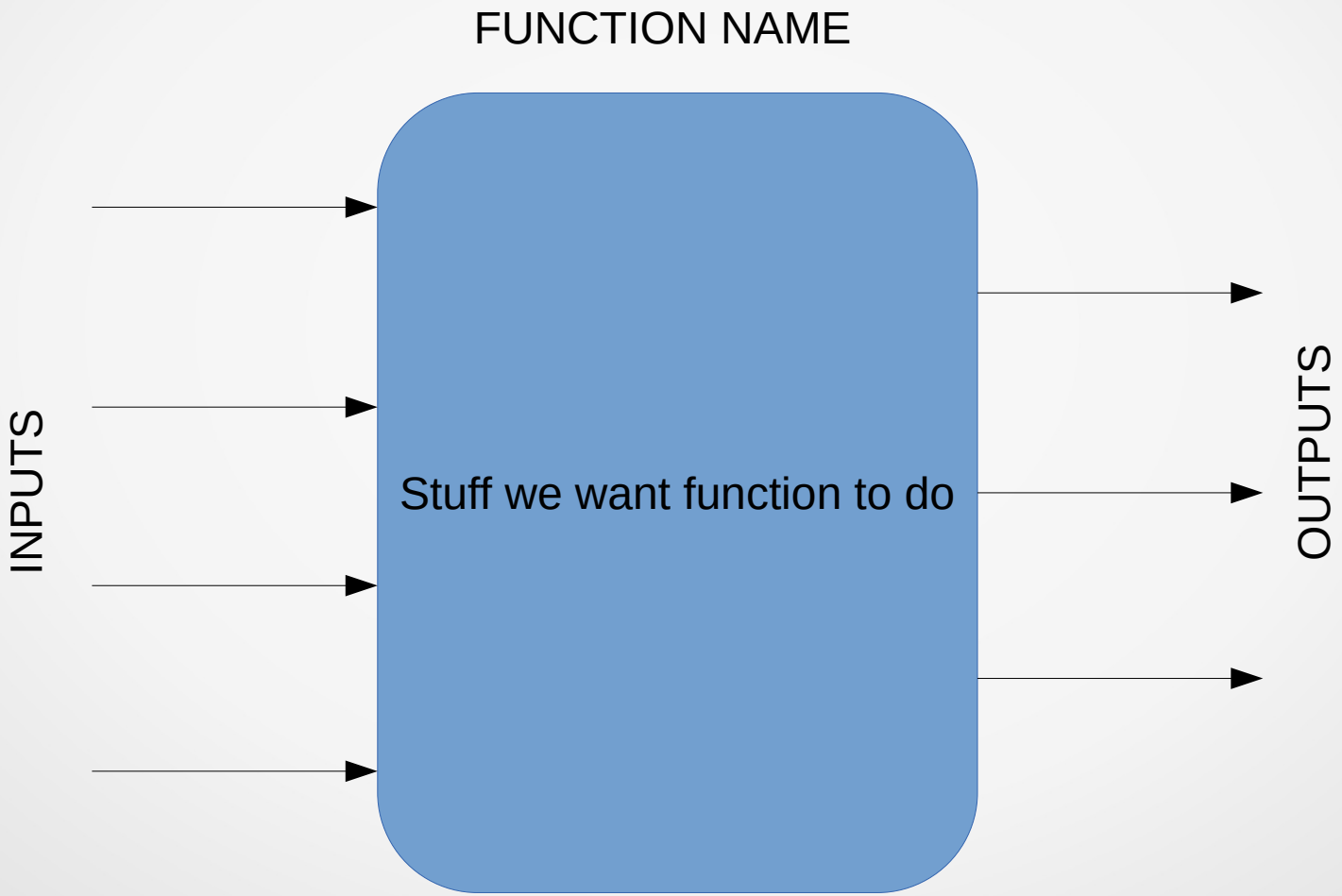
To prevent us having to write out the same sets of instructions time and again, we can wrap a set of instructions up in a *function*.

A function can take *inputs*, do something with them, and then return an *output*. The things it does are just sets of instructions.

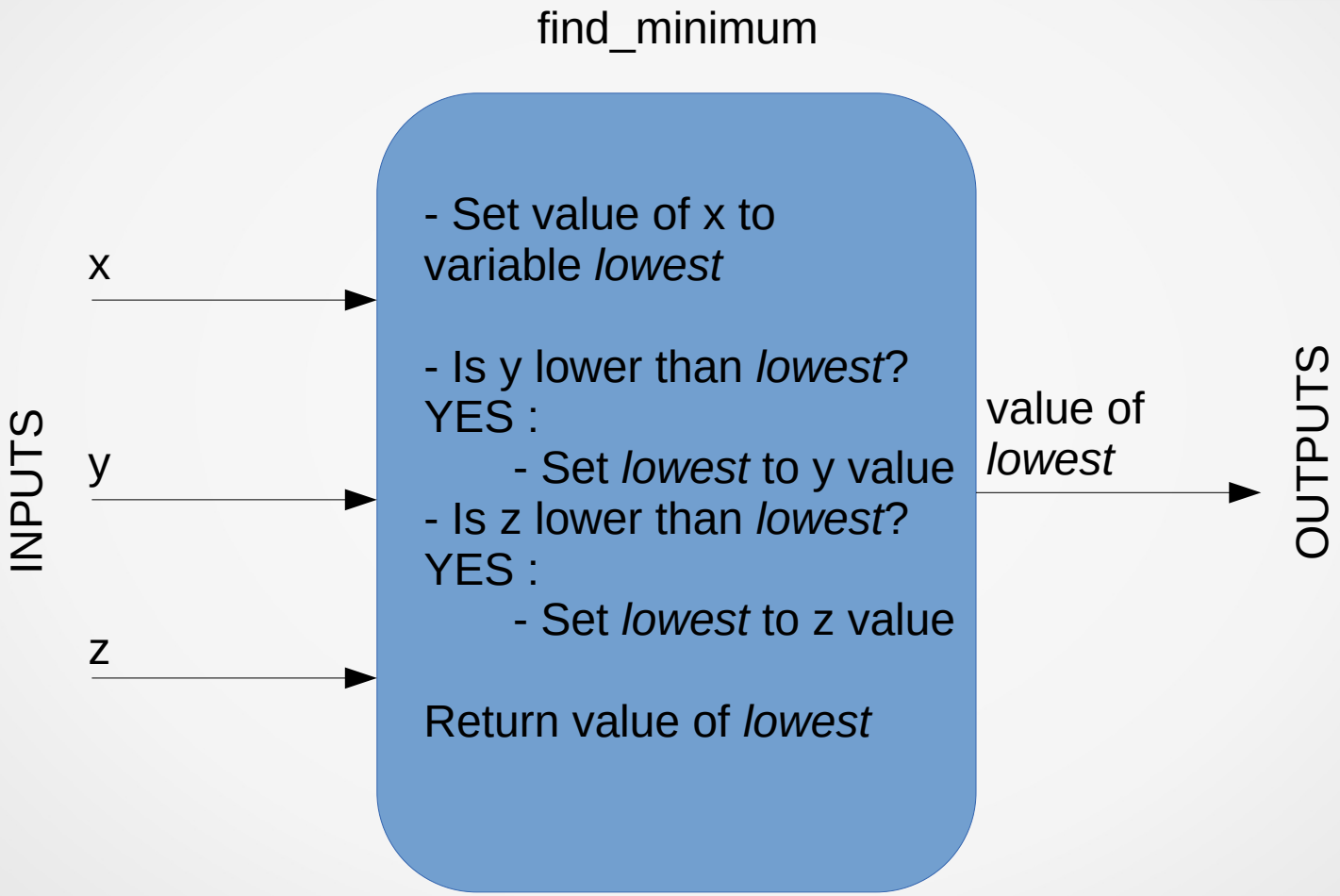
A function is defined by giving it a *name*, the *list of inputs* it should expect (if any), the things we want it to do, and the *outputs* it should *return* (if any).

The function doesn't do anything until it is called, however.

Functions



Functions



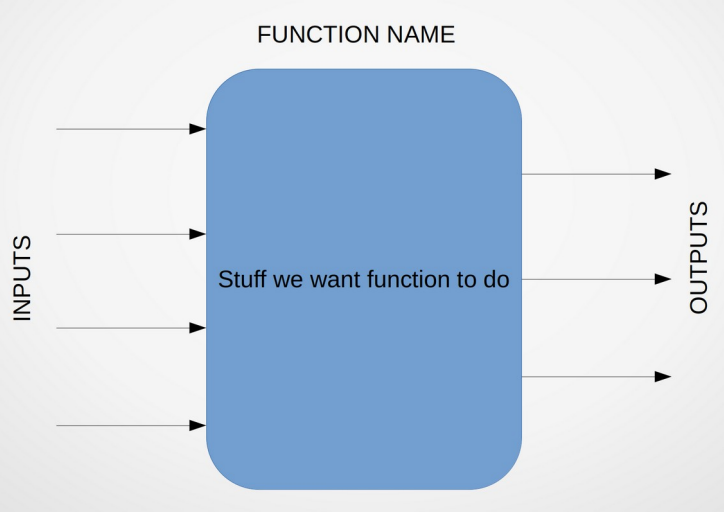
Example : find_minimum(3,7,2) would return the value 2

Writing Functions

As well as the many in-built functions and those in external libraries, we can also write our own functions.

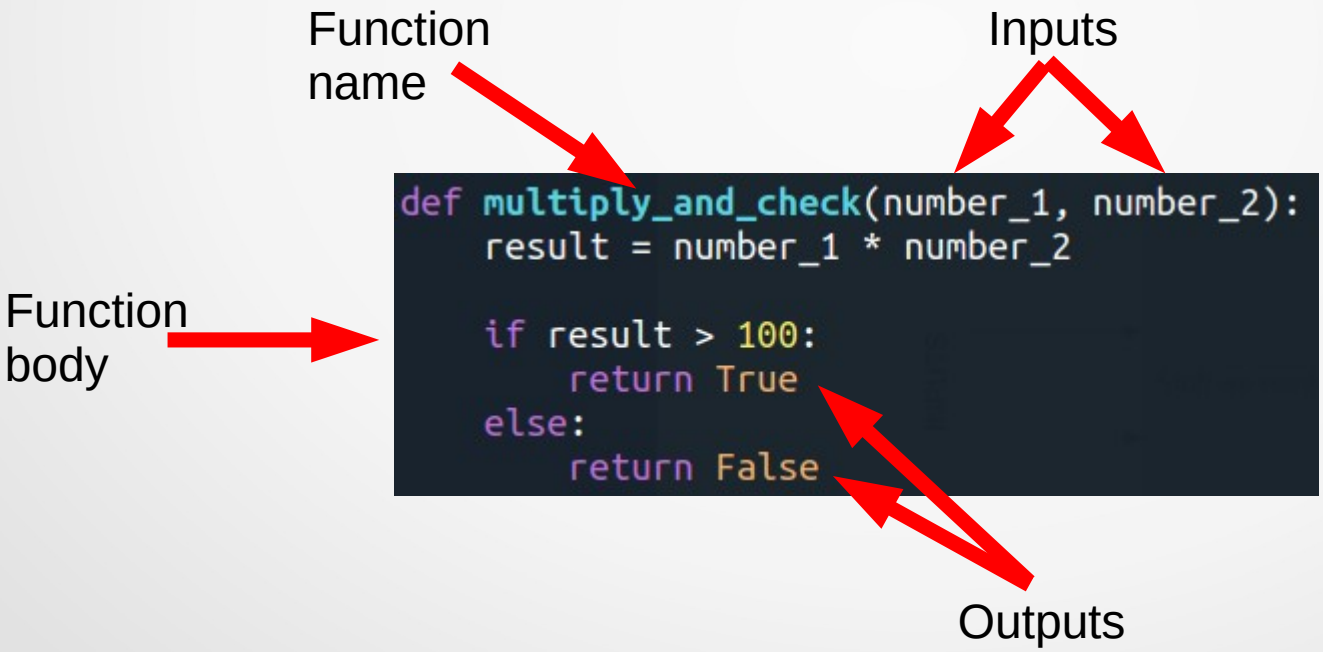
Indeed, most programs will (and should) have their own functions, so we don't have to repeat the same code multiple times.

To define a function we use the *def* command. We specify the function name, the input(s) to the function (if any), and the block of code representing the function's inner workings. We also usually need to return an output.



Function Example

Let's consider an example. Let's say we want to write a function that takes two numbers, multiplies them together, works out whether the result is higher than 100 and returns a Boolean to indicate this.



Calling a Function

Now we've written our function, we can call it at any time in our program. We need to ensure we give it the inputs it is expecting (in this case two inputs) and store the output of the function appropriately. **You must define the function before calling it!**

```
def multiply_and_check(number_1, number_2):
    result = number_1 * number_2

    if result > 100:
        return True
    else:
        return False

user_input_1 = int(input("Enter number 1 : "))
user_input_2 = int(input("Enter number 2 : "))

is_result_over_100 = multiply_and_check(user_input_1, user_input_2)

if is_result_over_100 == True:
    print ("I knew you'd think of big numbers!")
```

Note :
The names of the inputs when function is called don't have to match those used in the function definition

```
Enter number 1 : 480

Enter number 2 : 54
I knew you'd think of big numbers!
```

Comments

When writing programs, it is important to *comment* our code (leave notes explaining what various sections of code are doing) to make it easier for others to follow.

There are different ways to comment in Python :

```
# this is a comment, until we start a new line
# at which point we need another hashtag

"""This is also a comment, and will continue to be so until
the three speech marks close it again."""
```

(Re) Introducing….

Try, Except, Else, Finally

Try and Except

Sometimes, things in our program won't work as intended. For example, in our previous example, if the user inputs something other than an integer (e.g. some text that can't be converted to an Integer), Python will return an error and terminate the program when it runs this :

```
user_input_1 = int(input("Enter number 1 : "))
user_input_2 = int(input("Enter number 2 : "))
```

But that's messy. It would be far better to catch the error and deal with it. One way to do this is to use *try* and *except.*

Try and Except

A try and except statement is the most general way to handle errors in our Python code. It instructs Python to *try* a block of code, and if it runs into any problems, to run the block of code specified in *except*.

```
import random

def multiply_and_check(number_1, number_2):
    result = number_1 * number_2

    if result > 100:
        return True
    else:
        return False

try:
    user_input_1 = int(input("Enter number 1 : "))
except:
    print ("Sorry, you didn't input an integer")
    print ("We'll use a random integer instead")
    user_input_1 = random.randint(1,1000)
    print (f"Let's use {user_input_1}")

try:
    user_input_2 = int(input("Enter number 2 : "))
except:
    print ("Sorry, you didn't input an integer")
    print ("We'll use a random integer instead")
    user_input_2 = random.randint(1,1000)
    print (f"Let's use {user_input_2}")

is_result_over_100 = multiply_and_check(user_input_1, user_input_2)

if is_result_over_100 == True:
    print ("I knew you'd think of big numbers!")
```

```
Enter number 1 : Dan is
Sorry, you didn't input an integer
We'll use a random integer instead
Let's use 290

Enter number 2 : the best
Sorry, you didn't input an integer
We'll use a random integer instead
Let's use 608
I knew you'd think of big numbers!
```

Else and Finally

We can add to a try except statement by using *else* and *finally*.

```
try:
    # try this code
except:
    # run this if an error (exception) occurs
else:
    # run this only if you didn't get any exceptions
finally:
    # run this regardless of whether or not there were exceptions
```

Exercise 2

Let’s practice defining and using functions, and using exception handling.

Spend the next 40 minutes working through the Jupyter Notebook “python_prog_workbook_5.ipynb” in your groups.

Then take a 10 minute break before we rejoin.

(Re) Introducing….

Loops

Loops

Often, we need the computer to perform the same operation a number of times. It might be that :

- we need to do the same thing for a given number of iterations
- we need to continue to do something whilst a certain condition is true

We can use two different types of loop to achieve the above – *for* loops and *while* loops

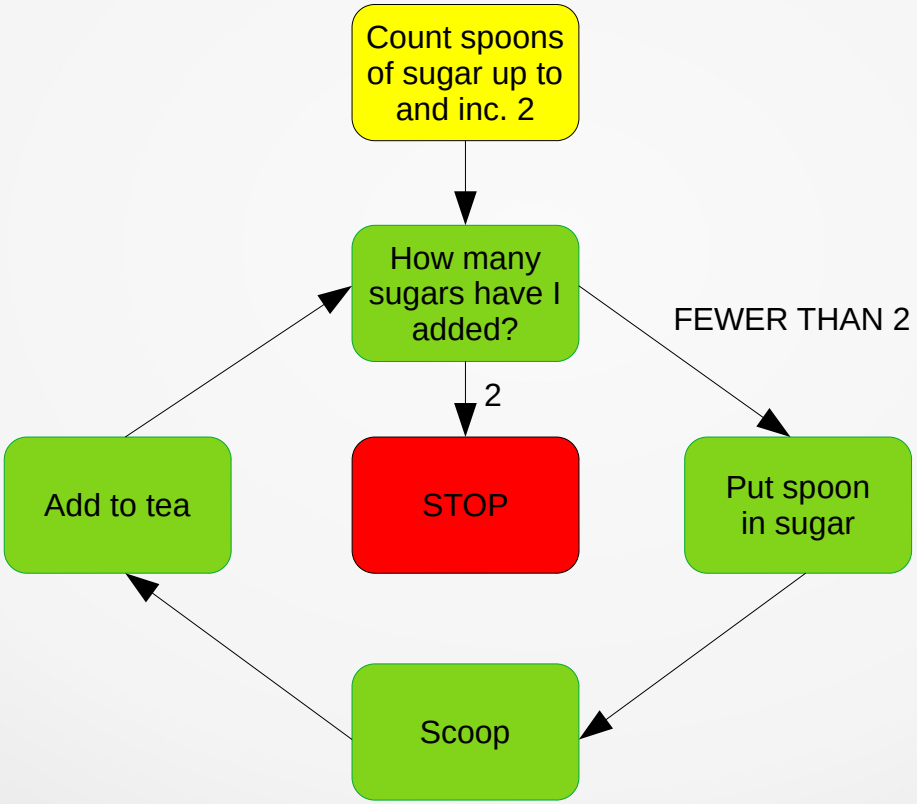
(Re) Introducing….

For Loops

1D

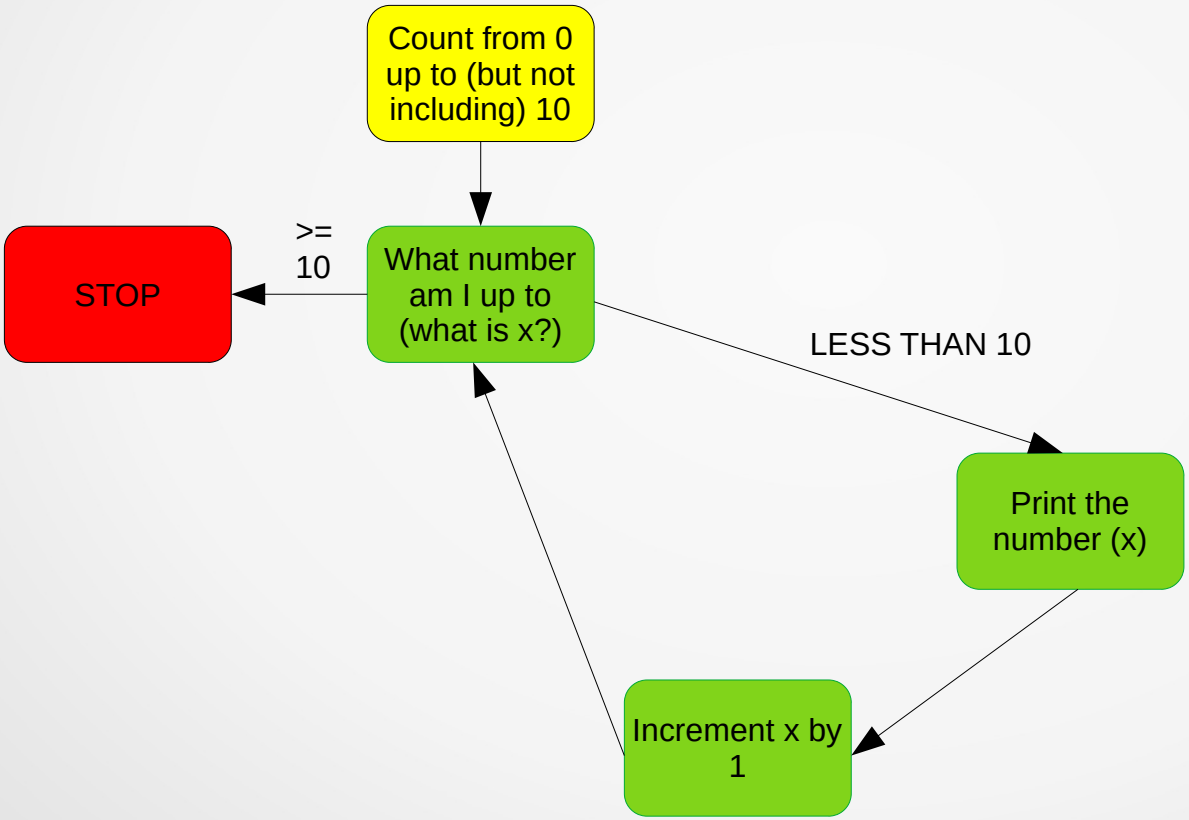
For Loops

Let’s say we want the computer to add two sugars to our cup of tea :



For Loops

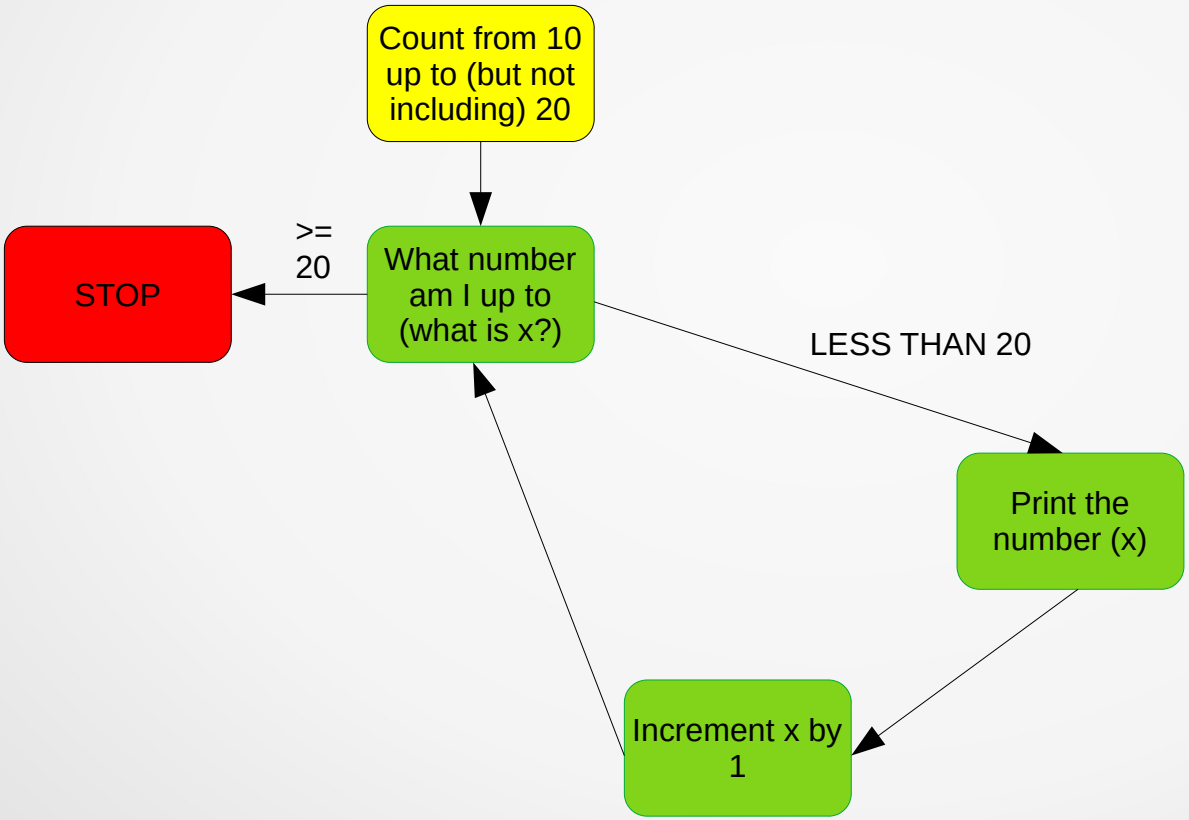
```
for x in range(10):
    print(x)
```




```
0
1
2
3
4
5
6
7
8
9
```

For Loops – Custom Range

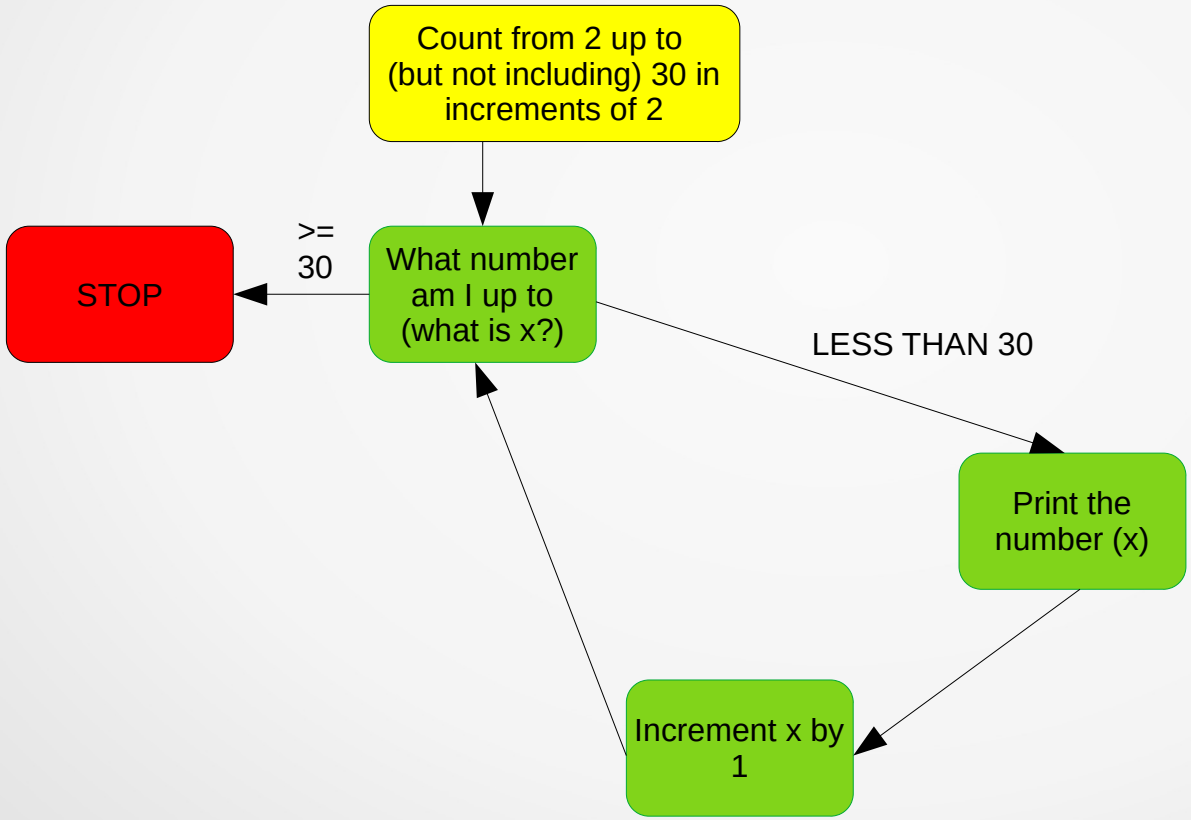
```
for x in range(10, 20):
    print(x)
```




```
10
11
12
13
14
15
16
17
18
19
```

For Loops – Custom Increment

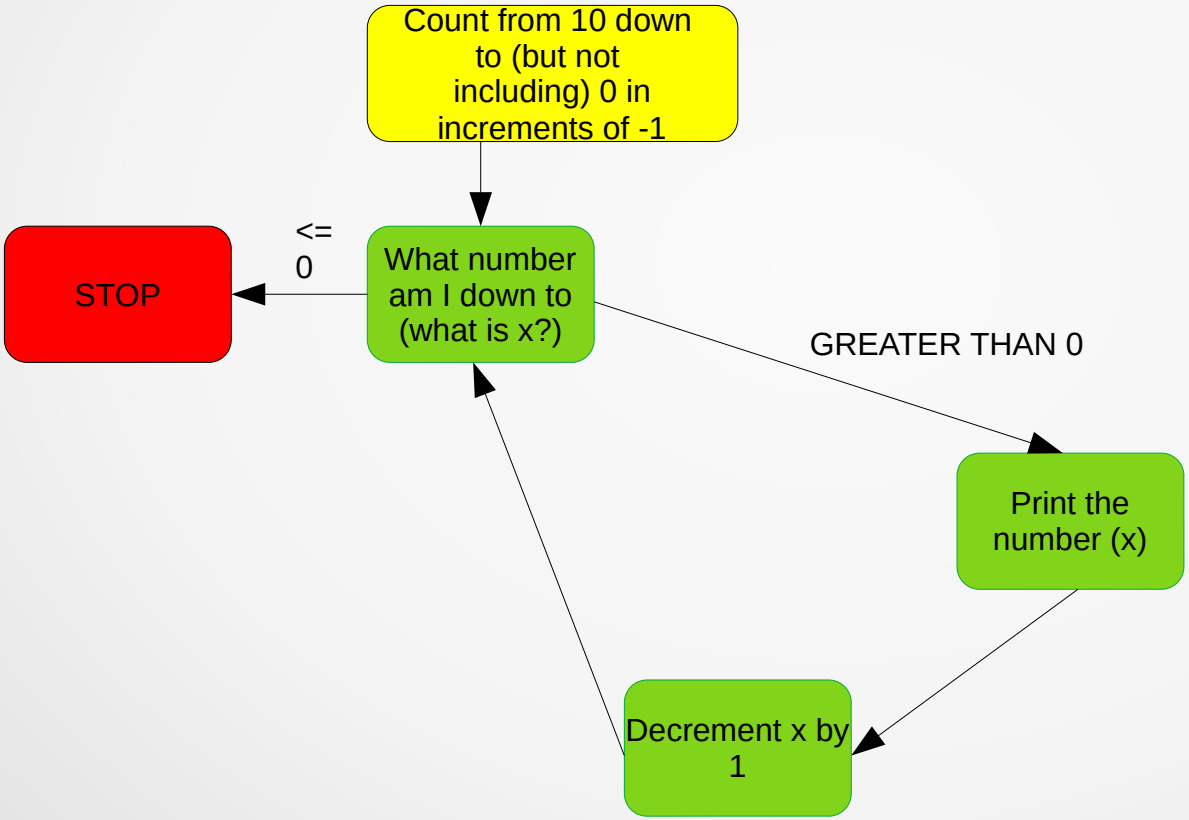
```
for x in range(2,30,2):
    print(x)
```




```
2
4
6
8
10
12
14
16
18
20
22
24
26
28
```

For Loops – Negative Increment (Decrement)

```
for x in range(10,0,-1):
    print(x)
```

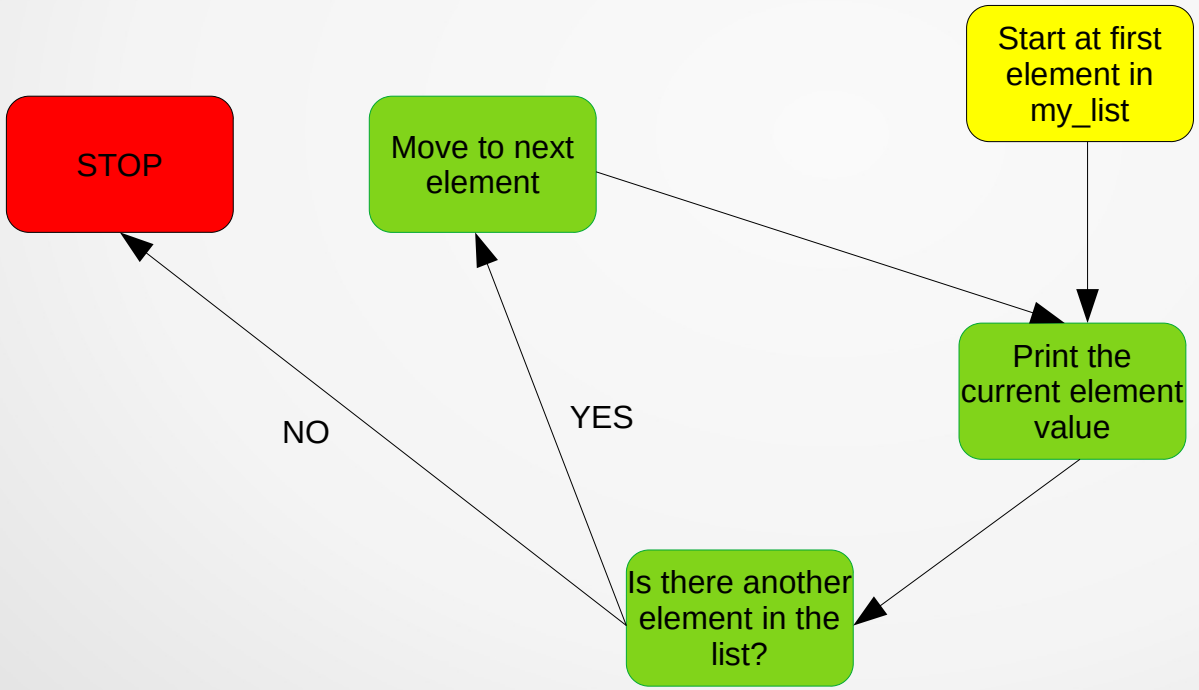



```
10
9
8
7
6
5
4
3
2
1
```

For Loops – Iterate through a List

```
my_list = [1,10,"Dan", True, "HSMA"]

for element in my_list:
    print(element)
```



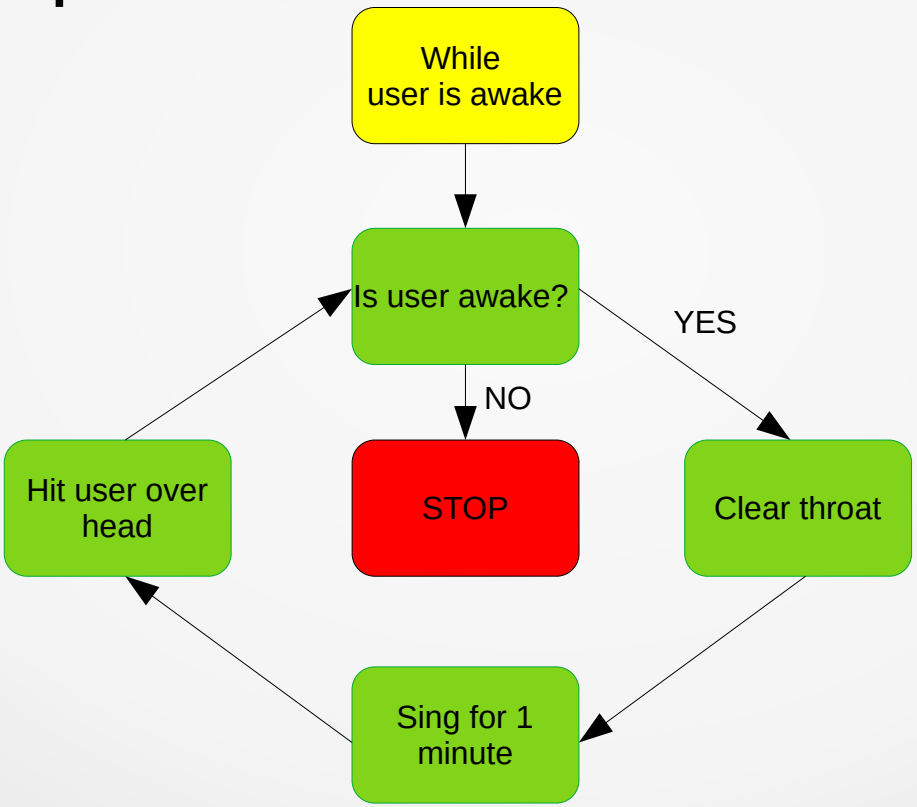

```
1
10
Dan
True
HSMA
```

(Re) Introducing….

While Loops

While Loops

Let’s say we want the computer to sing to us until we fall asleep :



Infinite Loops

Sometimes you can accidentally write an *infinite loop* – one that will never end. Anybody who used to play with BASIC in the old days will remember an example of an infinite loop :

10 PRINT “HELLO WORLD”
20 GOTO 10

More modern examples include setting up a while loop that can never stop :

```
x = 3

while (x >= 3):
    print(x)
    x += 1
```

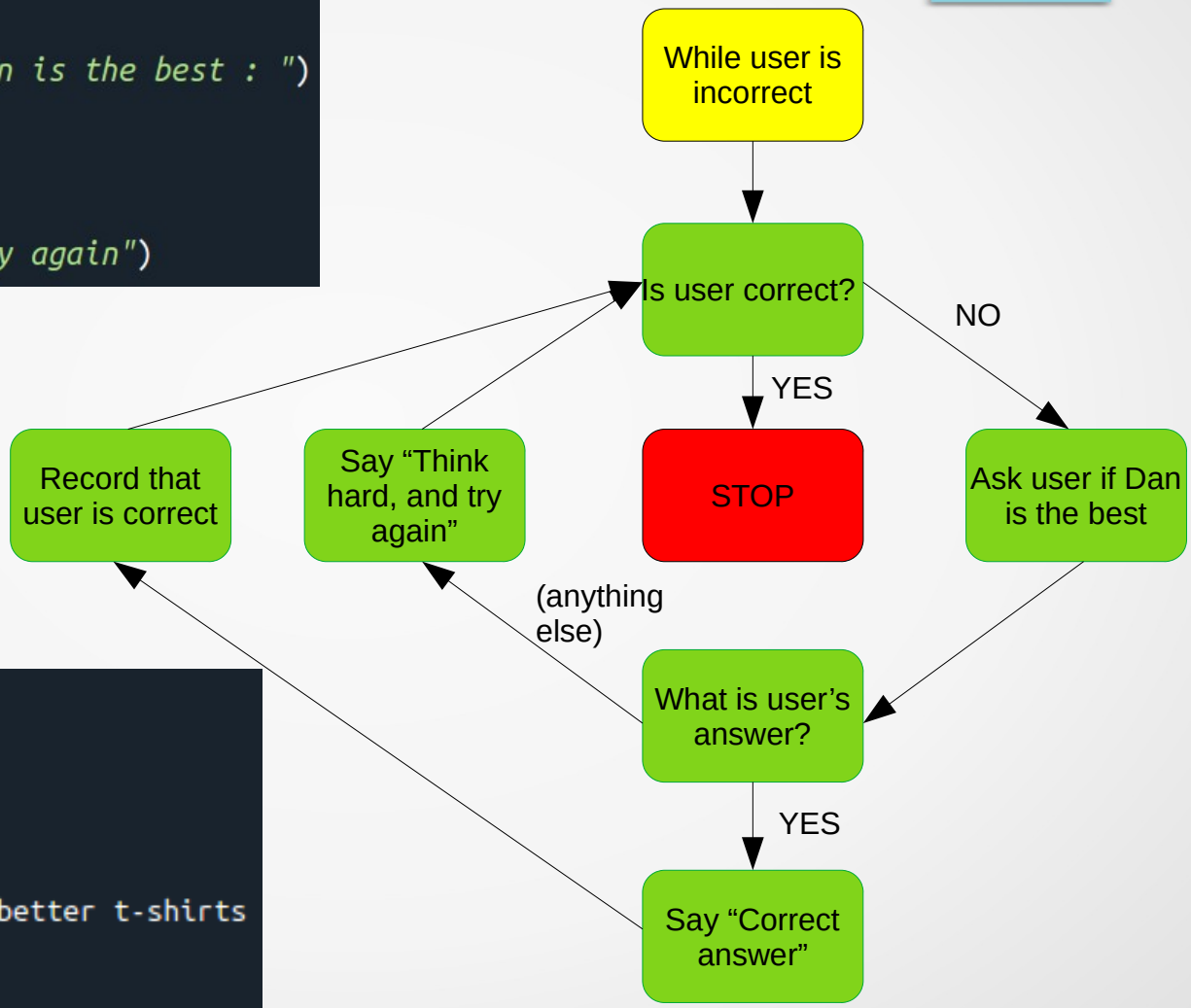
```
while True:
    print ("Dan is the best")
```

If this happens to you (and it will at some point) – don’t worry; just hit CTRL + C to interrupt and terminate the program (make sure you’re in the iPython console sub-window if you’re in Spyder). Sometimes, an infinite loop can be useful – for example you may want the user to stay in the system making inputs until they choose to close the system, at which point you can use a break statement.

While Loops

```
correct = False

while correct == False:
    answer = input("Type YES if Dan is the best : ")
    if answer == "YES":
        print ("Correct answer")
        correct = True
    else:
        print ("Think hard, and try again")
```




```
Type YES if Dan is the best : NO
Think hard, and try again

Type YES if Dan is the best : It's Mike
Think hard, and try again

Type YES if Dan is the best : He's got better t-shirts
Think hard, and try again

Type YES if Dan is the best : Sighs...
Think hard, and try again

Type YES if Dan is the best : YES
Correct answer
```

Breaking from a loop

Sometimes we want to break out of a loop mid-flow. For example, we may have a for loop, and want to break out of it when a condition has been met.

We can do this using the *break* command, which *immediately* breaks out of the loop and continues as though the for loop had completed.

Example :

```
total = 0

for x in range(10):
    total = total + x
    print (total)

    if total > 5:
        break
```

```
0
1
3
6
```

Exercise 1

Chalk Showers and Baths Inc (a division of Chalk Technologies – “building a brighter future”) has approached you to ask you to develop the software for a new automated shower system – the **C**halk **A**utomated **Sh**ower with **G**enius **R**esponses **A**nd **B**anter (CASH-GRAB) System.

In the new system, the user steps into the shower, and the shower asks them for their name. It then greets them with their name, saying “good morning”, “good afternoon” etc depending on the time of day. It then waits for the user to say “begin”, at which point the shower checks whether the user has a previous perfect temperature stored. If they do, it turns on at that temperature, and will remain at that temperature until the user says “too warm” or “too cold”.

If the user does not have a perfect temperature stored, the shower turns on at 30 Degrees C.
The shower continues to raise the water temperature by 2 degrees C every 5 seconds. If the user says “too warm”, the shower drops the temperature by 1 degree C every 5 seconds. If the user says “too cold”, the shower again starts raising the temperature by 2 degrees C every five seconds. If the user says “perfect”, then the shower holds the current temperature, and records it in a variable called *perfect_temp*, alongside their name in a variable called *user_name*, and stores both of them in a list called *user_preferences*. If the user had a previous perfect temperature, their new temperature is overwritten. Also, their calls of “too warm”, “too cold” and “perfect” will work as described above.

The shower continues to run until the user says “end shower”, at which point the shower says “Goodbye” followed by their name. In your groups, you have 45 minutes (+10 min break) to draw up an outline of the system. You should :
- Draw a diagram outlining how the system will work, including any conditional logic, loops and variable storage.
- Write a list of the variables that will be used in the system, along with the type of each variable, and a short (single sentence) description of what the variable is storing.

When we return, I’ll ask a few teams to share what they’ve done.

(Re) Introducing….

List Comprehensions

1D

List Comprehension

Python has a really useful feature called “List Comprehension” that allows us to easily create lists based off other lists using a single line of code.

Let’s consider an example. Let’s imagine we have a list of numbers, and we want to set up a second list containing all the numbers of the first list, but doubled.

Here’s one way we could do it :

```
list_a = [1,2,3,4,5]
list_b = []

for number in list_a:
    list_b.append(number*2)

print (list_b)
```

Or we could just use a list comprehension :

```
list_a = [1,2,3,4,5]
list_b = [x*2 for x in list_a]

print (list_b)
```

```
[2, 4, 6, 8, 10]
```

List Comprehension

We can also use conditional logic in a list comprehension, so that we form a list from another list where the elements meet certain criteria :

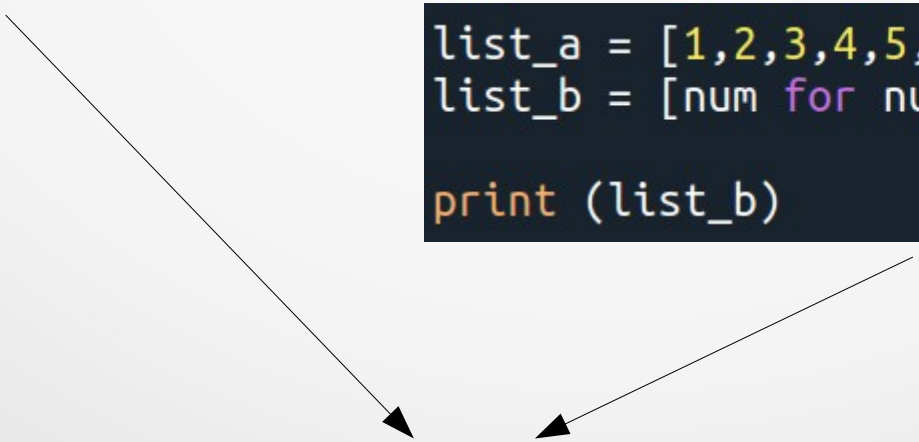
```
list_a = [1,2,3,4,5,6,7,8,9,10]
list_b = []

for number in list_a:
    if number % 2 == 0:
        list_b.append(number)

print (list_b)
```

```
list_a = [1,2,3,4,5,6,7,8,9,10]
list_b = [num for num in list_a if num % 2 == 0]

print (list_b)
```



```
[2, 4, 6, 8, 10]
```

Exercise 3

You've now learned about :
- For Loops
- While Loops
- Breaking from Loops

Switch to Jupyter Lab. Spend the next 30 minutes working through the Jupyter Notebook "python_prog_workbook_2". Remember, you can test if your code cell is working by running the cell (CTRL + Enter whilst in the cell)

Exercise 4

Spend 40 minutes working through “**python_prog_workbook_3**” in Jupyter Lab.