TEACH THE NATION TO CODE

A companion workbook to *Teach the Nation to Code*

RECAP

QA.com

Contents

- Introduction — 3
- Lab 1: First Python Exercises — 4
    - » Why do we need computers? — 4
    - » Why do we need variables? — 5
    - » How do variables work? — 6
    - » Inputs — 8
    - » Specificity — 9
    - » Data types — 10
    - » Strings — 11
    - » Integers and floats — 12
    - » Booleans — 13
    - » Control flow — 13
    - » Conditions — 14
    - » Operators — 14
    - » if: — 15
    - » else: — 16
    - » elif: — 16
    - » Iteration — 18
    - » for: — 19
    - » break — 19
    - » continue — 19
    - » while: — 20
  - › Lab 1 Recap: Python calculator — 22
- Lab 2: Flowcharts and Loops — 23
    - » Problems, Steps, Solutions — 23
    - » Code complexity — 24
    - » Flowcharts — 24
- Lab 3: Working from solutions — 30
    - » Solutions, steps, problems — 30
    - » Outputs — 30
- Summary — 33

Introduction

Teach The Nation To Code is a one-day series designed to get novice programmers with little or no experience to learn, write, and run code within the Python programming language.

TTNTC Recap is the second of three **Companions** to the Teach the Nation to Code programme, serving as a good indicator of the kinds of Topics and Exercises covered during a TTNTC Event. It is primarily aimed at those who are unable to attend an event.

Also, in the **TTNTC Companions** series are the following Handbooks:

- **TTNTC Rehearsal,** which contains pre-learning material with which one can use to prepare ahead of time for a TTNTC event, as well as post-learning material containing supplementary exercises to complete at leisure after attending an event;
- **TTNTC Revision**, which contains the answers to the Exercises contained in TTNTC Rehearsal and TTNTC Recap.

Combined, these Handbooks should allow you to learn, understand, and successfully implement Python code, even if you have never coded anything before.

Ensure that you have completed the Installing Python section of TTNTC Rehearsal before attempting the Exercises outlined in this Handbook.

Lab 1: First Python Exercises

Presumably, you're here because you're pretty new to coding in general (or, at least, new to Python) – and that's okay, because we all have to start somewhere. The good thing about Python is that it's incredibly lightweight, so you can be up and running with your first scripts in a matter of minutes.

These first few exercises will give you a basic understanding of the general way in which Python works. You can work through them at whatever pace you like. The idea here is to get you up and running – not to become a Python Wizard instantly.

It is presumed that you have completed the **Pre-Event Lab** from **TTNTC Rehearsal** in preparation for these Exercises.

Why do we need computers?

Most people think that computers solve problems. They are present in our everyday lives and are perhaps the single biggest driver of humanity's technological advancement – so it makes sense to think this is true.

However – computers **don't** solve problems. We do.

The most important phrase you will ever learn from Teach The Nation To Code is this:

a computer is a stupid machine

They're stupid machines because, no matter what, computers will only ever do exactly what we ask them to do – no more, and no less.

However, they are useful for two reasons: speed and accuracy. For these two reasons, they are excellent for helping to solve problems.

Let's try to calculate the following:

176 times 325 is...

If we try to calculate something as difficult as this, without a computer, it will take a long time to do it, and we might end up getting the answer wrong.

Feed this into a calculator, however, and we know two things:

1. we'll **always** get the correct answer
2. we'll **immediately** get the correct answer

Exercise 0: Speed and accuracy
As a quick recap, we will re-familiarise ourselves with writing and executing programs in Python.

Output the answer to this question to the console.

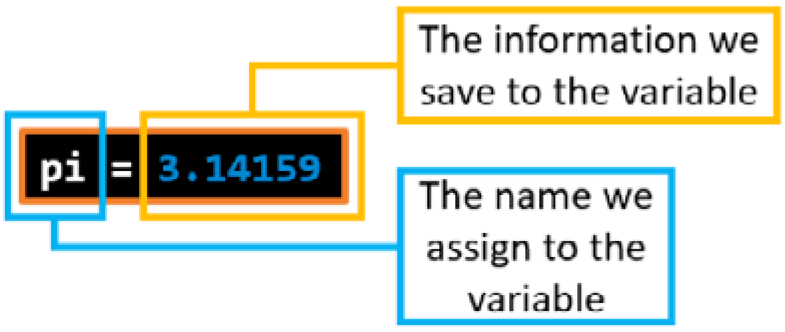
Create a file named QuickCalc.py for this task.

We'll come back to why Python acts as a calculator in Exercise 3.

Why do we need variables?

Let's say we have a complicated piece of information that is difficult to store, such as a full sentence or a large number.

If we want to reference this information multiple times within a program, it makes sense to store this information in an easy-to-access location. This way, rather than needing to write the same information over and over again, we can just point to the location we've stored that information in instead. That's all a variable is.



```
pi = 3.14159
```

The information we save to the variable
The name we assign to the variable

Exercise 1: Assignment
Store the phrase **hello my friends** in a variable, then output it to the console.

Create a file named HelloAssignment.py for this task.

How do variables work?

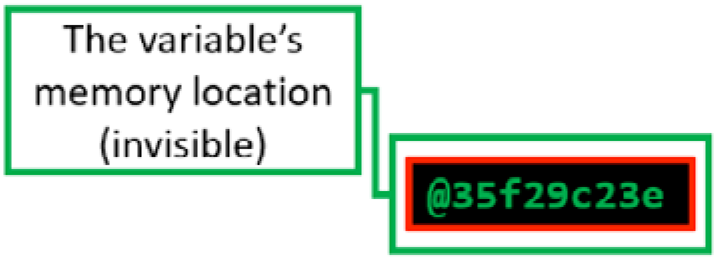
We know that a variable holds information we want to save. The place that we save this information to is somewhere in the computer's memory.

The good thing about saving it in the computer's memory is that it doesn't matter exactly where the information is saved to.

The random area of memory where we store the information has two things – a **name** that we call it, and an **address** for where it is – attached to it.

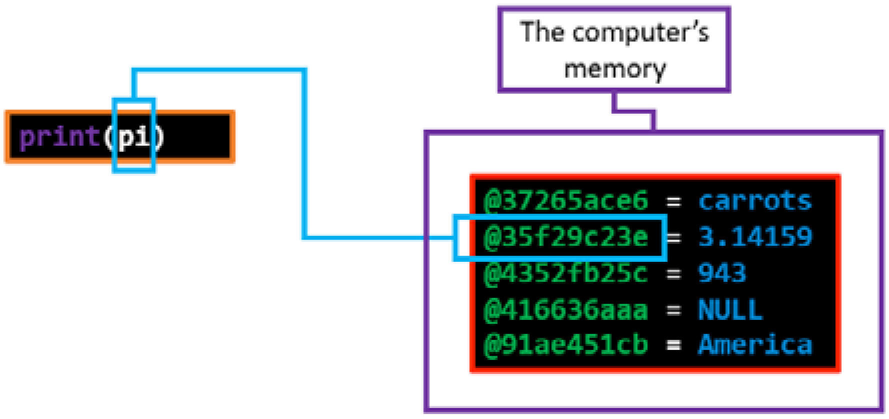
The address indicates its physical location in the computer's memory, and the name which we give the variable is used to point to the address whenever we want to get that information.

What it's doing under the hood is storing the value of pi in some random physical location in the computer's memory:



Rather than needing to remember the address of that memory location, we just assign that location the name **pi**.

pi now "points to" that location, so whenever you say you need the value of pi in a program, you just use the variable called **pi** instead:



```
print(pi)
```

```
@37265ace6 = carrots
@35f29c23e = 3.14159
@4352fb25c = 943
@416636aaa = NULL
@91ae451cb = America
```

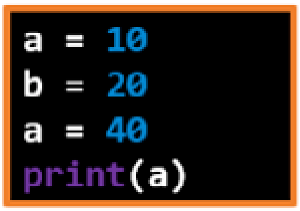
The **name** of the variable links to the **address**, making the information we need accessible at any time – so, above, the result of **print(pi)** is that we output **3.14159** to the screen.

Exercise 2: Reassignment
So why is a variable named a variable?

Because that information – the thing we save – is **able** to be **varied**.

It can be used, changed, and then used again without any problems at all.

The place where it is saved doesn't change, and the name of the variable doesn't change – but the information can, if we need it to.



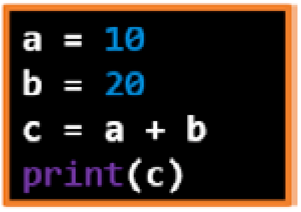
```
a = 10
b = 20
a = 40
print(a)
```

Here, we have two variables named **a** and **b**. Replicate this setup within your code. Before running the code, write down what you think the value of a will be once the program has finished being executed.

Create a file called Reassignment.py for this task.

Exercise 3: Expressions
Python, like other programming languages, also acts as a calculator, thanks to the way in which operators are used.

Create a file called Expressions.py for this task.



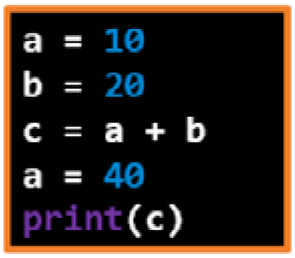
```
a = 10
b = 20
c = a + b
print(c)
```

We've now changed the code we had for Exercise 2 around a bit, with the important inclusion of a new variable called c. Replicate this setup within your code.

Before running the code, write down what you think will be outputted to the screen from the print statement.

Exercise 4: Execution

Python, like most other programming languages, executes from **top-to-bottom**. Essentially, once the compiler – the bit that does the running of code – is done with a line, it will move onto the next one.



```
a = 10
b = 20
c = a + b
a = 40
print(c)
```

Create a file called Execution.py for this task.
Here, we have re-assigned **a** to 40 after calculating the result of **c**. Replicate this setup in your code.

Before running the code, write down what you think the values of the three variables will be at every line of code.

Inputs

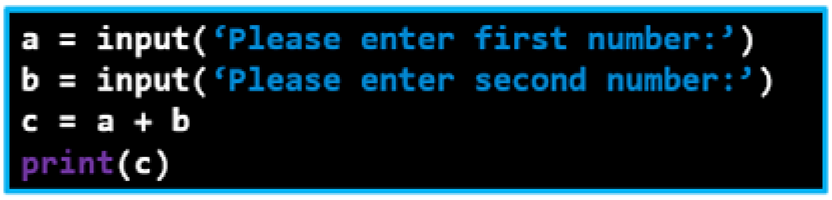
These initial programs all run based on data that you write into it as the programmer.

Each of the values which we make variables for are single values which cannot be accessed and changed by the person using the program.

If we were to give our programs to another person to use, they won’t really do much in practice. The values we put into it are hard-coded, rather than given to us by a user.

Exercise 5: input()

We can change our code from being programmer-centric (with hard-coded values) into a user-centric one by using the **input()** function:



```
a = input(‘Please enter first number:’)
b = input(‘Please enter second number:’)
c = a + b
print(c)
```

Create a file named Inputs.py for this task.

Replicate the above setup in your code and run it.

You should find that the program will now wait for the user to input a number.

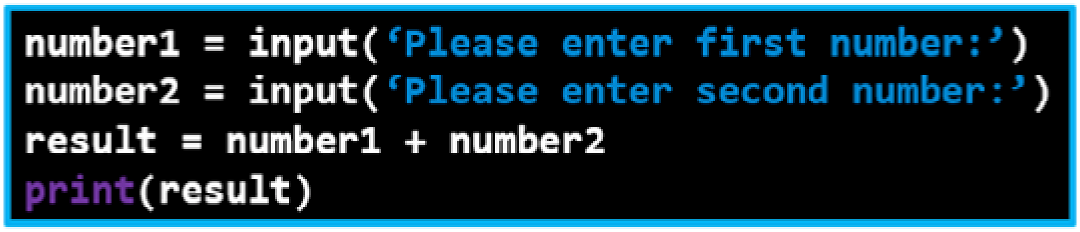
What the code is doing under the hood is assigning the variables **a** and **b** to anything that the user enters.

What do you think will happen if we attempt to enter “two” for one of the variables?

What might happen if we enter “#” as one of the variables?

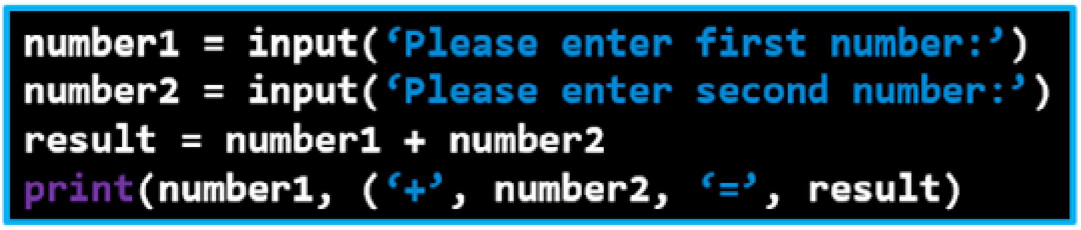
Specificity

There are many ways that we can improve this code.
The first, and easiest, way of doing so, is by using useful variable names. This may seem like a waste of time now, but in larger projects it will prove to be invaluable – both to you, as the developer, and to anybody you share your code with, as somebody trying to read it:



```
number1 = input(‘Please enter first number:’)
number2 = input(‘Please enter second number:’)
result = number1 + number2
print(result)
```

We can also make the result that we show to the user easier to understand:



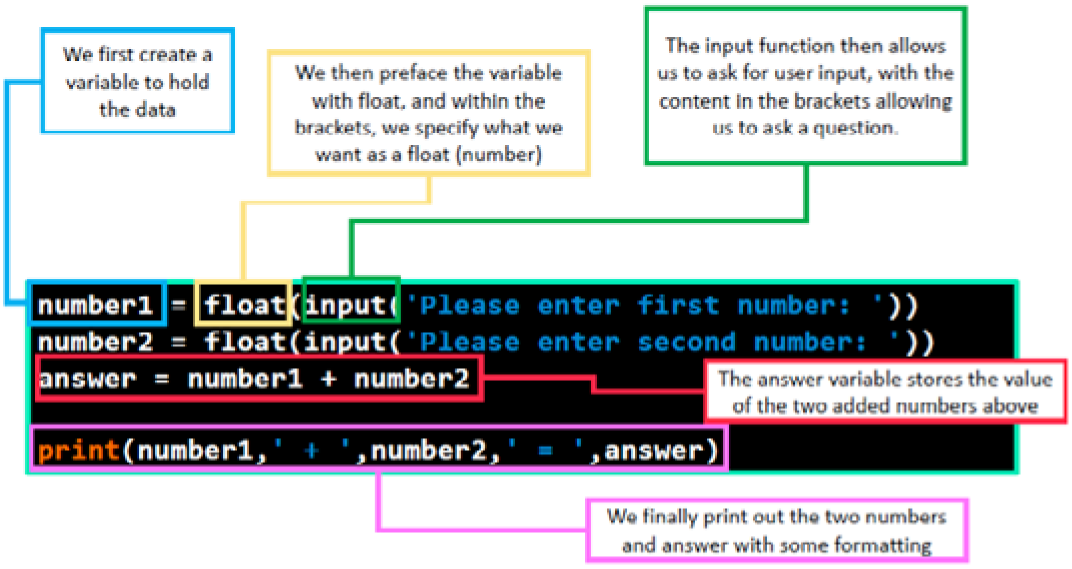
```
number1 = input(‘Please enter first number:’)
number2 = input(‘Please enter second number:’)
result = number1 + number2
print(number1, (‘+’, number2, ‘=’, result)
```

Exercise 6: Casting
Even if we begin to make the code easier to understand, it will still fail if we give it a number that is written as a word, or something that isn’t a number at all, as an input.

We should make it clear to users that when we ask them to enter a number, we want this number to be a digit.

We can do this by changing the **type** of the data that the code will accept. (We'll move onto **types** later in this Handbook.)

In the below code, we **cast** the number we get from the user's input to something useful:



```
number1 = float(input('Please enter first number: '))
number2 = float(input('Please enter second number: '))
answer = number1 + number2

print(number1,' + ',number2,' = ',answer)
```

We first create a variable to hold the data
We then preface the variable with float, and within the brackets, we specify what we want as a float (number)
The input function then allows us to ask for user input, with the content in the brackets allowing us to ask a question.
The answer variable stores the value of the two added numbers above
We finally print out the two numbers and answer with some formatting

The **float()** function **casts** the information we feed into it into a data type called **float**.

Replicate this setup in your code to see it at work. Create a file named Casting.py for this task.

What do you think a **float** is?

Change all the **float()** functions to **int()** functions instead. What do you think an **int** is?

Data types

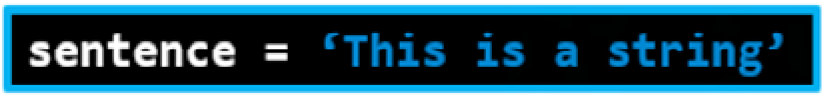
Information comes in various forms. We can categorise the kinds of data we need to use in our code by classifying them into data types.

A data type is an attribute of data that tells us how a programmer intends to use the data.

Data types define the operations that can be done on the data, the meaning of the data, and the way values of that type can be stored.

We have already seen a few data types in action: **float** and **int**.

We have also used one data type without knowing its name:



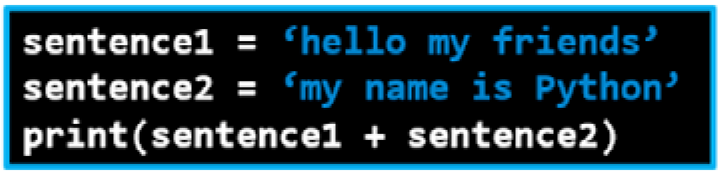
```
sentence = ‘This is a string’
```

Strings

A **String** is declared by encasing any other text in a set of inverted commas. Essentially, a String is a block of text.

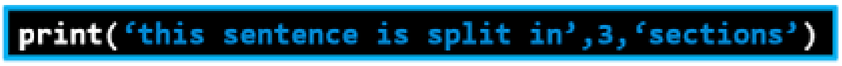
There are a number of things to bear in mind when using Strings in Python:

- It is known as a **str**
- **input()** defaults the return value to a String – hence why we cast it
- We can **concatenate** Strings together by using **+**:



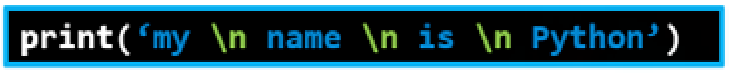
```
sentence1 = ‘hello my friends’
sentence2 = ‘my name is Python’
print(sentence1 + sentence2)
```

- We can combine Strings with other data types by using **,**:



```
print(‘this sentence is split in’,3,‘sections’)
```

- We can write Strings over multiple lines by using **\n**:



```
print(‘my \n name \n is \n Python’)
```

Exercise 7: Playing around with Strings

Create a file named Strings.py for this task.

Write some code that uses some of the String-concepts we have already covered.

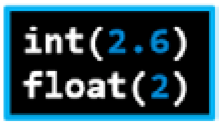
Integers and floats

These are the two main **numeric data types** in Python.

The difference between these two data types and Strings is that we can calculate with them, as we've already seen.

The key difference between **float** and **int** is that the former uses decimal points, while the latter does not.

Exercise 7: Playing around with numeric data types
Create a file named Numeric.py for this task.



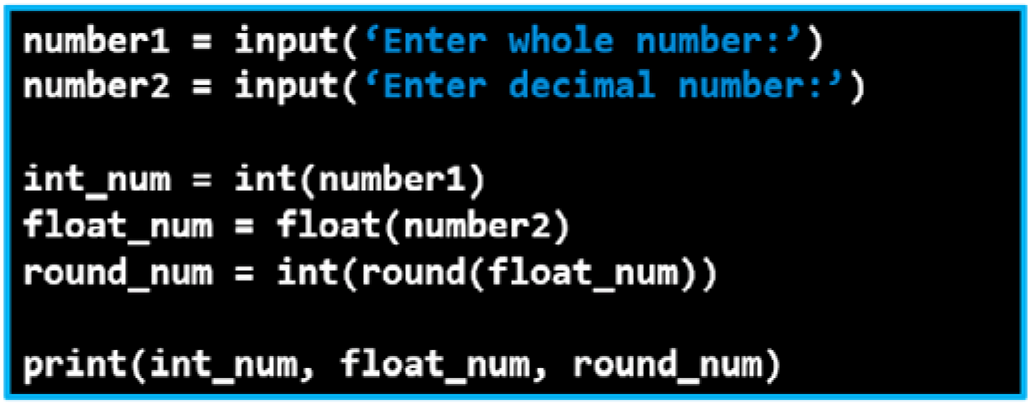
```
int(2.6)
float(2)
```

Using these two code snippets, print out the results of casting these numbers.

What do you notice about them?

Exercise 8: Rounding
Create a file named Rounding.py for this task.

Replicate the code setup below:



```
number1 = input(‘Enter whole number:’)
number2 = input(‘Enter decimal number:’)

int_num = int(number1)
float_num = float(number2)
round_num = int(round(float_num))

print(int_num, float_num, round_num)
```

Before running this code, write down what you think the output will be.

Booleans

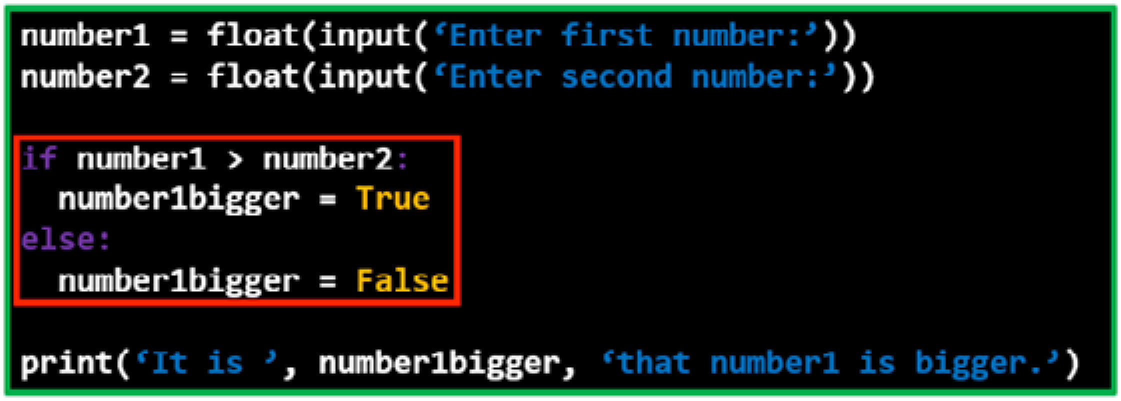
A **Boolean** is a data type usually treated as non-numeric. It can only hold one of two values: **true** or **false**. In Python it is symbolised by the term **bool**. Booleans are used for logical tests:

- Is something bigger or smaller?
- Has something been completed?
- Has a condition been fulfilled?

They are used in tandem with conditional statements.

Exercise 9: Working with Booleans
Create a file named Booleans.py for this task.

Replicate the code below in your file:



```
number1 = float(input('Enter first number:'))
number2 = float(input('Enter second number:'))

if number1 > number2:
  number1bigger = True
else:
  number1bigger = False

print('It is ', number1bigger, 'that number1 is bigger.')
```

We can see here that we have conditions, which, when fulfilled, will set **number1bigger** to either be **True** or **False**.

What do you think the code is doing in the middle section highlighted in red?

Control flow

Before we look further into **conditionals**, it is important to consider the way in which code executes.

Earlier, we touched on how Python, like every other major programming language, executes from top to bottom. Essentially, once a line of code has completed, the compiler will always move down the program by one line.

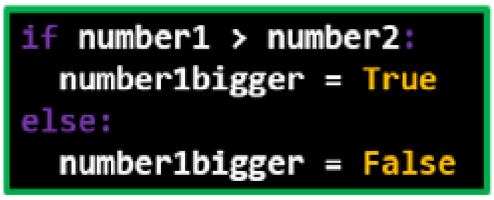
Control flow is simply the order that lines of code are executed in. By using statements which manipulate the flow of a program, the programmer can determine which section of code is run in a program at any time.

It is important to master this art.

It is critical to understand, in any program, which blocks of code are going to execute, and when. This is important not only for writing your own programs, but for reading and understanding code that you haven't written.

Conditions

As seen in Exercise 9, we can write some code which only runs if a condition is true:



```
if number1 > number2:
  number1bigger = True
else:
  number1bigger = False
```

For this set of statements, we know that **number1bigger** will only ever be **True** if **number1** is bigger than **number2**. We define this by using **<** in this case.

Operators

There are several other operators which can be used within conditional statements, some of which you may recognise from mathematics:

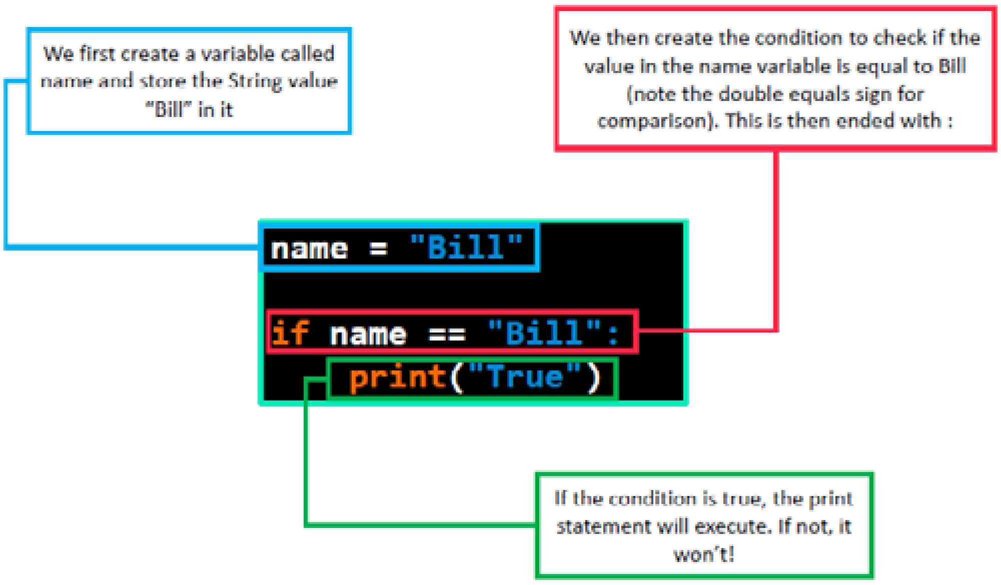
Operator	**description**
>	Greater than
<	Less than
==	Equal to
!=	Not equal to
>=	Greater than or equal to
<=	Less than or equal to

To check if a condition is True, we can these operators alongside conditional statements.

if:

The most well-used conditional statement is the **if-statement.** If a condition evaluates to being True, then any code encased within the if-statement is run.

In Python, it looks like this:

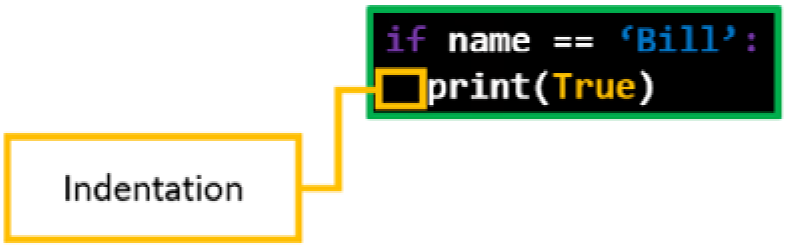


```
name = "Bill"

if name == "Bill":
    print("True")
```

We first create a variable called name and store the String value "Bill" in it
We then create the condition to check if the value in the name variable is equal to Bill (note the double equals sign for comparison). This is then ended with :
If the condition is true, the print statement will execute. If not, it won't!

If, for whatever reason, we assigned a variable called **name** to something that was not 'Bill', then the print statement would never run - the compiler would simply skip over that line of code.



```
if name == 'Bill':
    print(True)
```

As you can see, the code which is accessed within the if-statement is indented. The reason for this is extremely important:

```
in Python, you must indent the block of
  code after a conditional statement
```

An indentation is 4 spaces/1 tab.

By laying out code in this way, both the programmer and the compiler is clearly able to see which blocks of code execute in which situations.

Exercise 10: Practicing conditionals
Create a file named Conditionals.py for this task.

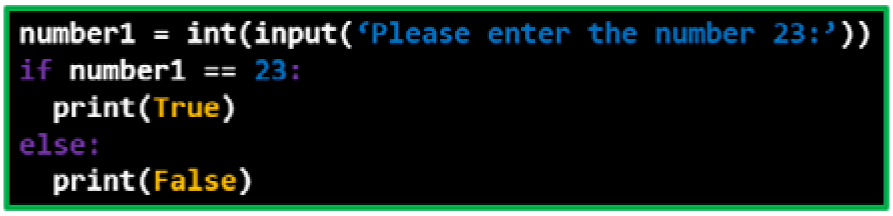
Write a program which asks for an input of a mark, then checks to see if the mark is greater than 65. If so, print out **Pass**. If not, print **Fail**.

There are multiple ways of achieving this – but for now, do not use the else statement which you might have noticed we used earlier.

else:

else is a catch-all conditional statement. It is used to check any cases which an if-statement does not check for.

For instance, let's look at the following code:



```
number1 = int(input('Please enter the number 23:'))
if number1 == 23:
  print(True)
else:
  print(False)
```

In this code, we ask the user to enter the number 23 – and, in **any** case where the number 23 is not entered, the else-statement will activate and print **False** to the screen.

Exercise 11: if/else
Use the file from Exercise 10 for this task.

Update your program to use the **else**-statement to solve the problem

elif:

We don't always need our programs to check if every if statement evaluates to True. In a lot of cases, only one outcome can be True.

This issue can be solved by using an else-if-statement. The **else-if-statement** is defined in Python as **elif**.

By using an elif statement, you can make your code more efficient.

Exercise 12: if/elif/else

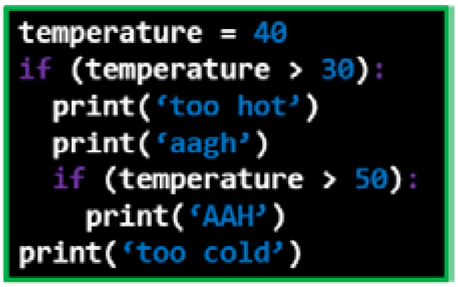
Use the file from Exercise 10 for this task.

Update your program to account for multiple grade boundaries – e.g. a **Merit**, a **Distinction**, a **Pass** and a **Fail** grade.

Exercise 13: Indentation

Create a file called TemperatureGauge.py for this task.

Replicate the following code in your file:



```
temperature = 40
if (temperature > 30):
  print('too hot')
  print('aagh')
  if (temperature > 50):
    print('AAH')
print('too cold')
```

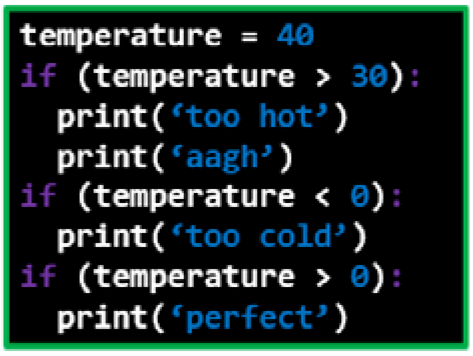
What will happen if we set **temperature** to be 40? 60? 20?

What problems are we seeing occur in the code? How can we fix it?

Exercise 14: Many ifs

Use the file from Exercise 13 for this task.

Replicate the following code in your file:



```
temperature = 40
if (temperature > 30):
  print('too hot')
  print('aagh')
if (temperature < 0):
  print('too cold')
if (temperature > 0):
  print('perfect')
```

What will happen if we set temperature to 20? 40? -10?

Exercise 15: And an else
Use the file from Exercise 13 for this task.

Edit the final if-statement to be an else-statement.

What is this implementation doing that makes it different from the several if-statements we used in Exercise 14?

Exercise 16: Improved if/elif/else
Use the file from Exercise 13 for this task.

Edit the code to use a set of if-, elif-, and else-statements.

Why is this implementation better than using the several if-statements written in Exercise 14? Think about how each block of your code links together.

Why is this implementation better than the if and if-else statements used in Exercise 15?

Iteration

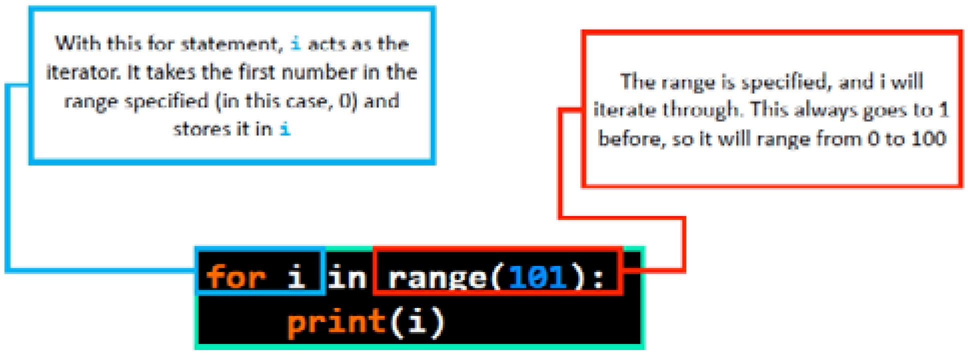
Iterations are often used in conjunction with, or instead of, conditional statements.

Iterations are loops. By writing an iteration, we are writing something which we expect to loop through several times.

In Python, there are two types of iteration which we shall focus on for now – the **for-loop** and the **while-loop.**

for:

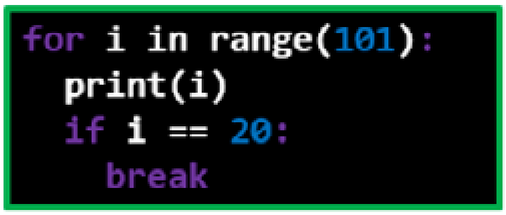
A for loop is used to loop through code a specified number of times.
The loop count will always start at 0 and finish one less than the number specified:


With this for statement, i acts as the iterator. It takes the first number in the range specified (in this case, 0) and stores it in i
The range is specified, and i will iterate through. This always goes to 1 before, so it will range from 0 to 100

```
for i in range(101):
    print(i)
```

break

If we want to stop a loop before the specified end, we can use the **break-statement:**

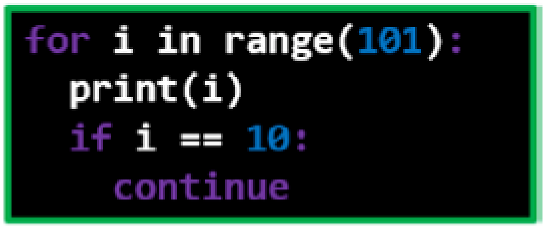


```
for i in range(101):
  print(i)
  if i == 20:
    break
```

The above code will loop through all numbers up to 101, as before – but as soon as the loop reaches **20**, the code will immediately finish executing – we **break** out of the loop.

continue

Sometimes we might want to skip a particular iteration within a loop. For this, we can use the **continue-statement**:



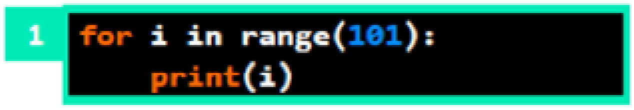
```
for i in range(101):
  print(i)
  if i == 10:
    continue
```

The above code will also loop through all numbers up to 101 as before.

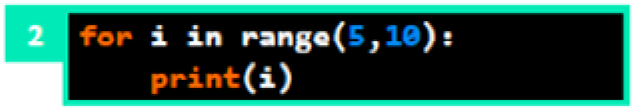
As soon as the loop reaches **10**, the code will simply pretend that this iteration through the code never happened and **continue** without doing anything.

Exercise 17: Looping through
Create three different Loop.py files for this task (name them accordingly).

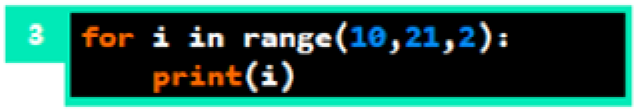
Set up each file according to the below code:



1
```
for i in range(101):
    print(i)
```



2
```
for i in range(5,10):
    print(i)
```



3
```
for i in range(10,21,2):
    print(i)
```

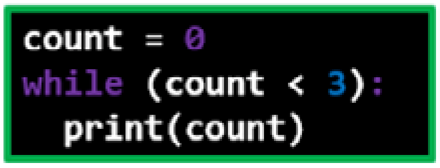
What is the output for each of these loops?

What is the variable i acting as in these examples?

while:

For-loops will repeat a specified number of times. However, **while-loops** will repeat until some condition we specify is false.

Exercise 18: Zero
Create a file named Zero.py for this task.



```
count = 0
while (count < 3):
  print(count)
```

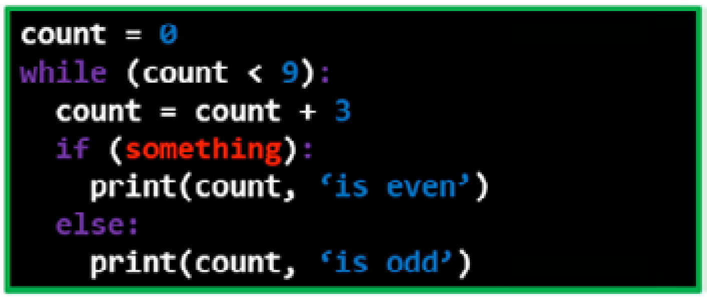
Replicate the above code in this file. What do you expect it to do if it is run?

Rewrite the code so that it counts up to 3.

Exercise 19: Odd/Even

Create a file named OddEven.py for this task.

Replicate the following code within your file:



```
count = 0
while (count < 9):
  count = count + 3
  if (something):
    print(count, ‘is even’)
  else:
    print(count, ‘is odd’)
```

Rewrite **something** so that for every number it loops through, the program outputs whether the number is odd or even.

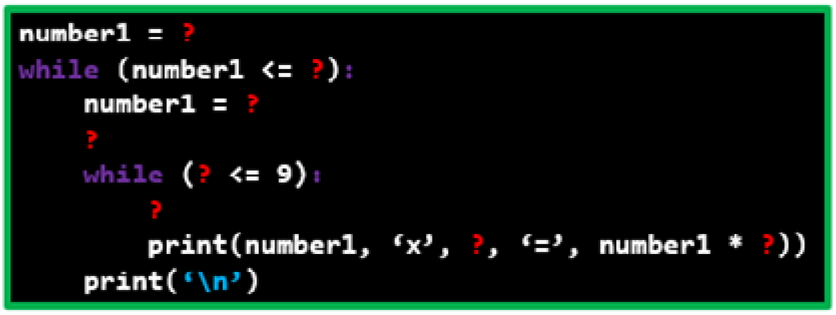
Exercise 20: Times Tables

Create a file named TimesTables.py for this task.

Using the following code as a baseline, write a program which writes the times tables up to 10.

The first output should read **1*1=1**.

The last output should read **10*10=100.**



```
number1 = ?
while (number1 <= ?):
    number1 = ?
    ?
    while (? <= 9):
        ?
        print(number1, ‘x’, ?, ‘=’, number1 * ?))
    print(‘\n’)
```

Lab 1 Recap: Python calculator

As an amalgamation of everything you have learned so far, your next challenge is to create a calculator in Python.

You can accomplish this task in any way that you wish – but there are some specifications which you will need to bear in mind:

- Your calculator must perform addition, subtraction, multiplication or division, depending on user input.
- You must allow users to input the two numbers they wish to calculate together.
- You must give your users the option to begin the program again (hint: make a variable which evaluates to **True** which the rest of the program relies on).

Lab 2: Flowcharts and Loops

This Lab will build upon the foundations which we have covered so far, by taking you though several iterative exercises that incorporate the Python syntax which you have learned so far.

The most important thing to remember from this Lab is this:

logic > understanding syntax

If you have a good understanding of how to logically break down problems into small steps, then you will be able to apply that logic to **any programming language.**

Developing your logic in Python will allow you to apply that logic to any other language that you want to learn. It may take some time to learn the new **syntax** of that language, but the underlying **logic** will remain the same.

Problems, Steps, Solutions

Perhaps the clearest way to break down a **problem** into a solution is to divide that problem into **steps**.

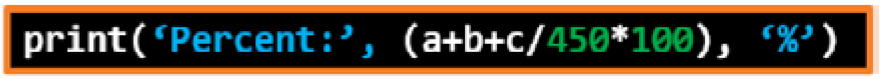
Programmers across the world use several techniques to break down the bigger problems they encounter. Here, we will try to do something similar.

Let's try to complete a fairly complex exercise together first.

Exercise 0: Grade summariser
Write a program which asks a student for their **marks** in three subjects: **chemistry, physics, and maths**. The program should print the **total** marks across all three subjects, a **percentage** score out of a maximum of 150 per subject, and, if the percentage is above 60%, tell the student that they have passed their exams.

Code complexity

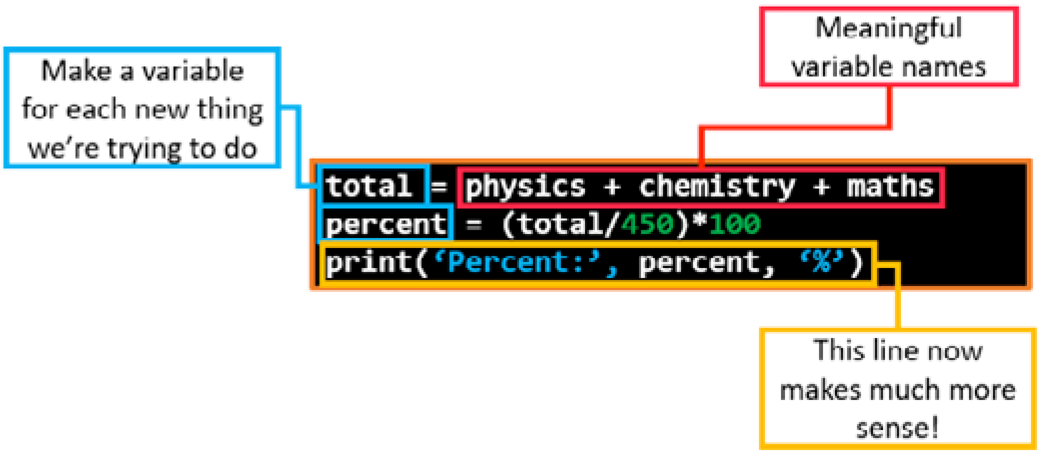
For the previous Exercise, you may have typed out something like this for each grade:



```
print('Percent:', (a+b+c/450*100), '%')
```

This is difficult to read. If you hand your code to somebody else, they may have a hard time understanding what you're trying to do.

We can fix this problem in a few ways:



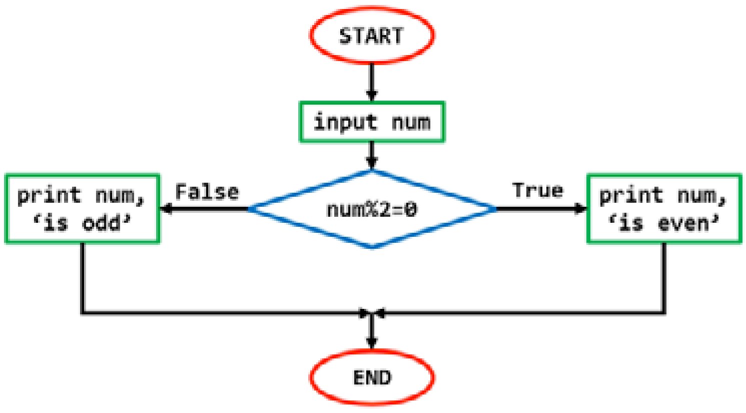
```
total = physics + chemistry + maths
percent = (total/450)*100
print('Percent:', percent, '%')
```

Make a variable for each new thing we're trying to do
Meaningful variable names
This line now makes much more sense!

Generally, it is better to do one thing in one line of code. The more you attempt to convey in the space of a line, the greater the complexity of your code.

Flowcharts

A good way of breaking down a problem into clear steps is to use a **flowchart**. Let's make one which checks to see if an inputted number is even:

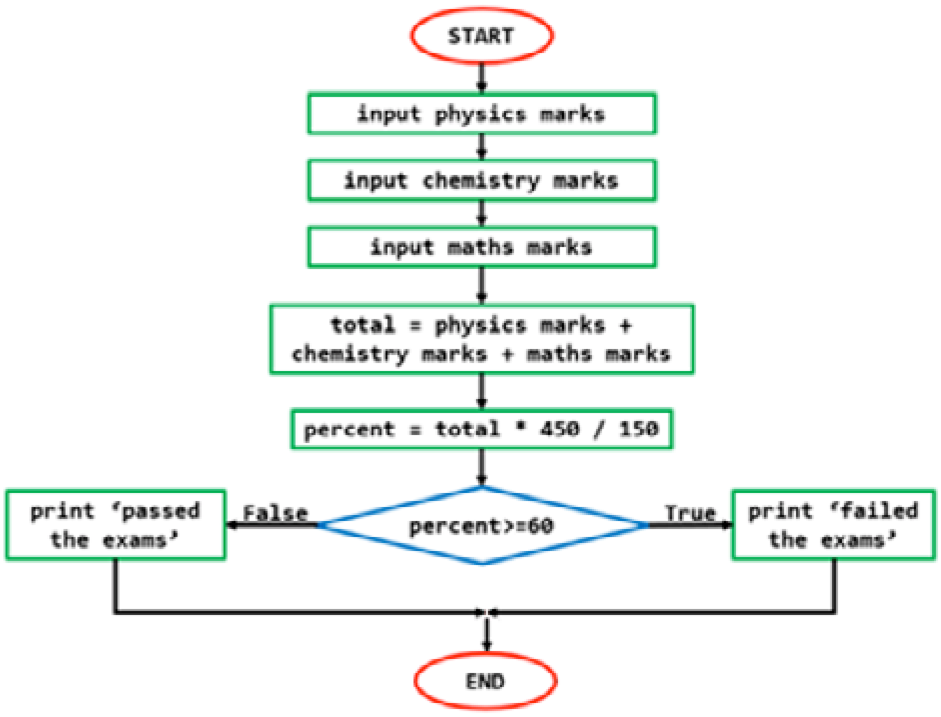


The flowchart operators are relatively easy to understand, and we can clearly see the steps which we would need to replicate in our code in order to fulfil requirements:

- **red circles** signify the start and end points for our program;
- **green rectangles** are single lines of code;
- **blue diamonds** represent conditionals and iterators;
- **black arrows** denote control flow;
- and **black labels** show situations where the control flow changes

We shall be using these for the duration of this Lab.

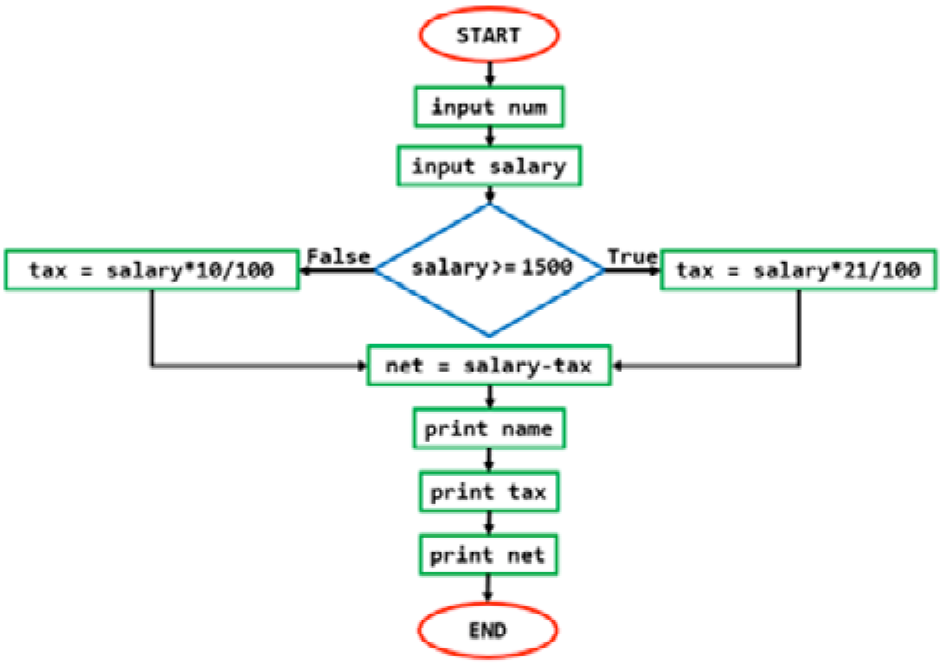
Exercise 1: Grade Summariser – with flowcharts
Here's a flowchart for our Grade Summariser:



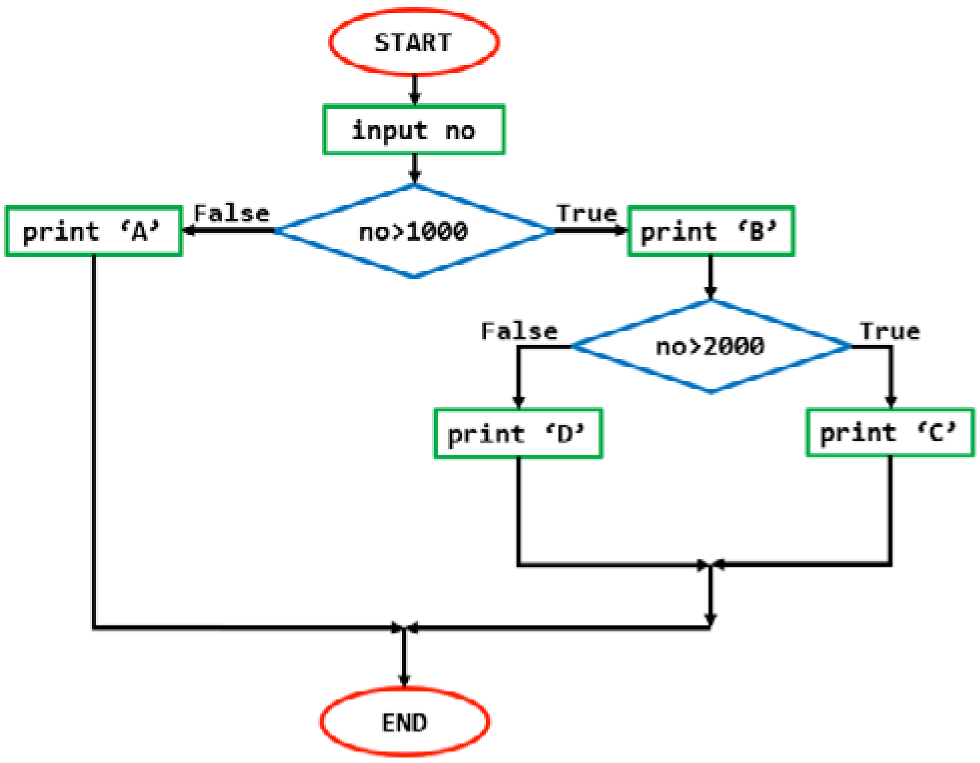
Re-write your program so that it follows each of these steps.

Exercise 2: Salary calculator

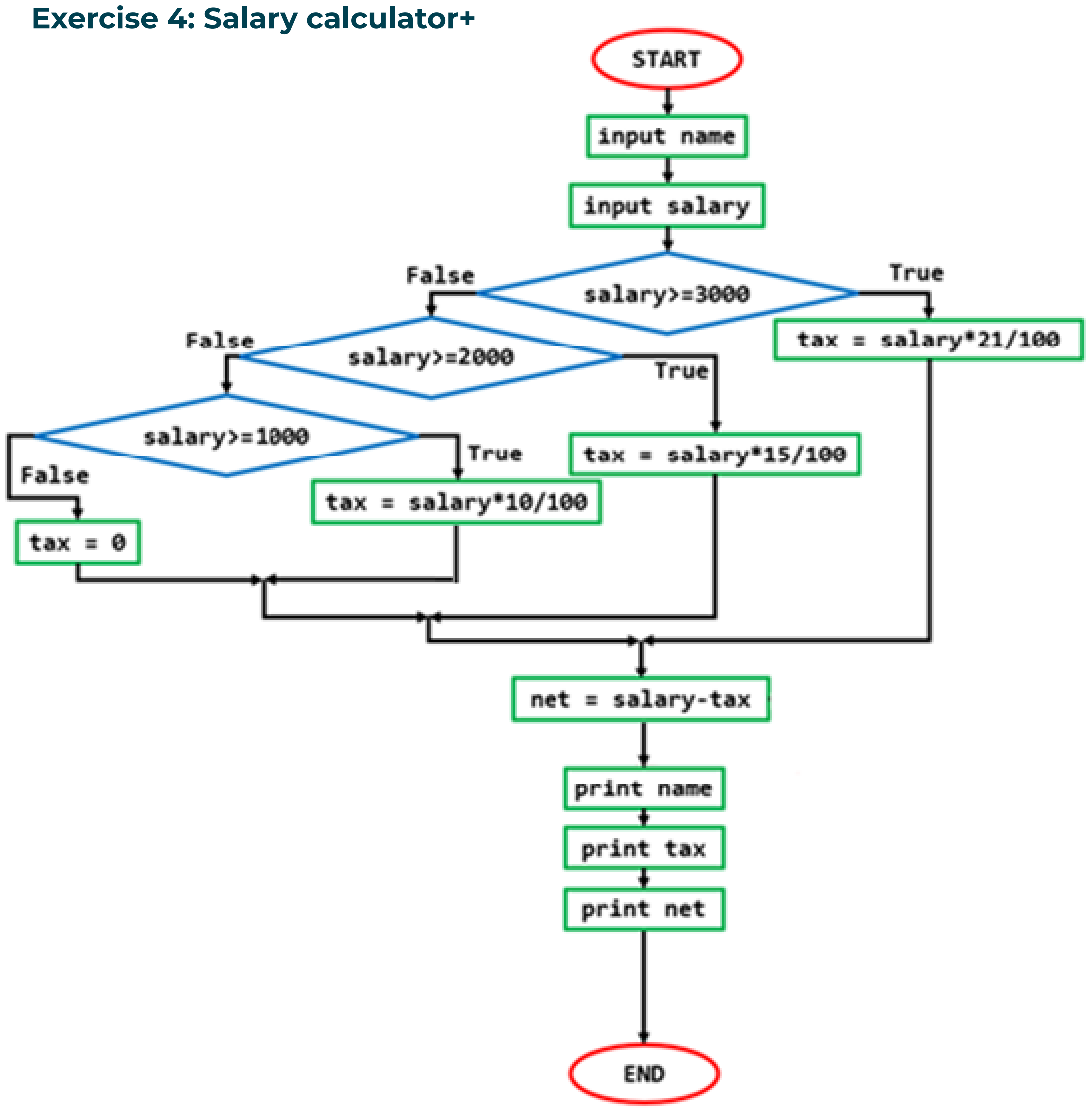
Use the following flowchart to write a program which calculates a salary:



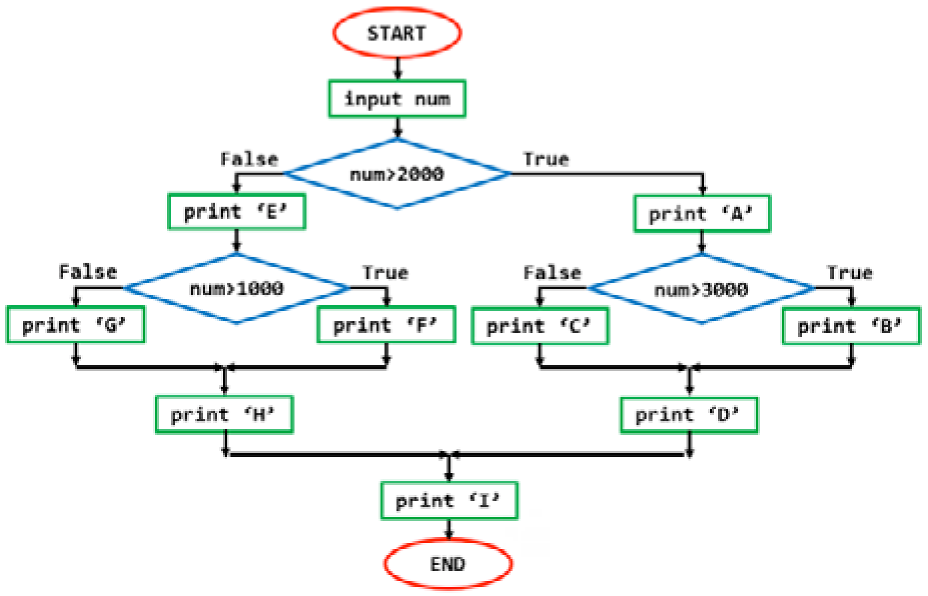
Exercise 3: Alphabetiser



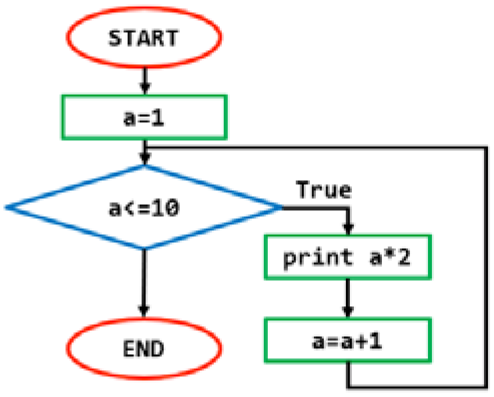
Exercise 4: Salary calculator+



Exercise 5: Alphabetiser+

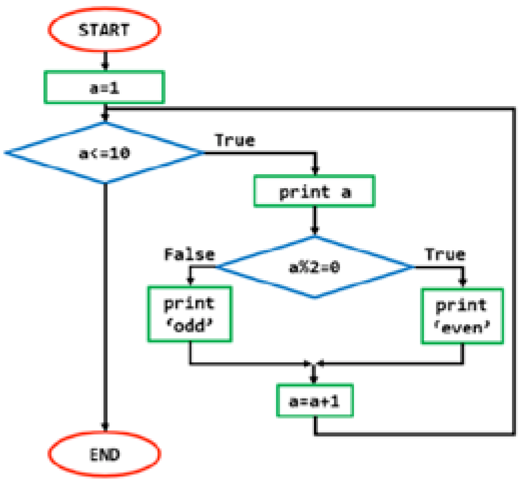


Exercise 6: Flowing while:

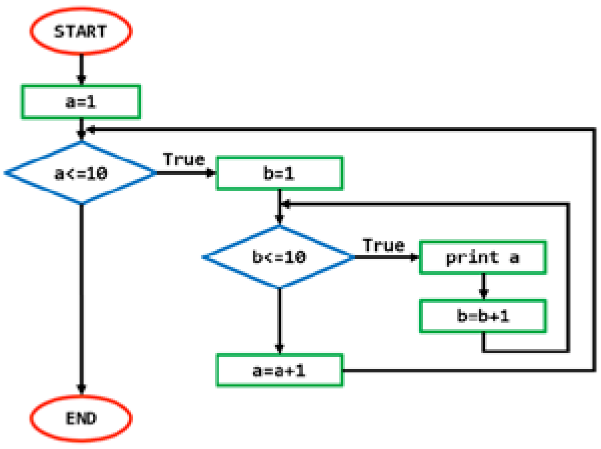


As a while loop will only evaluate while a condition remains True, we have no need for a False condition in our flowchart.

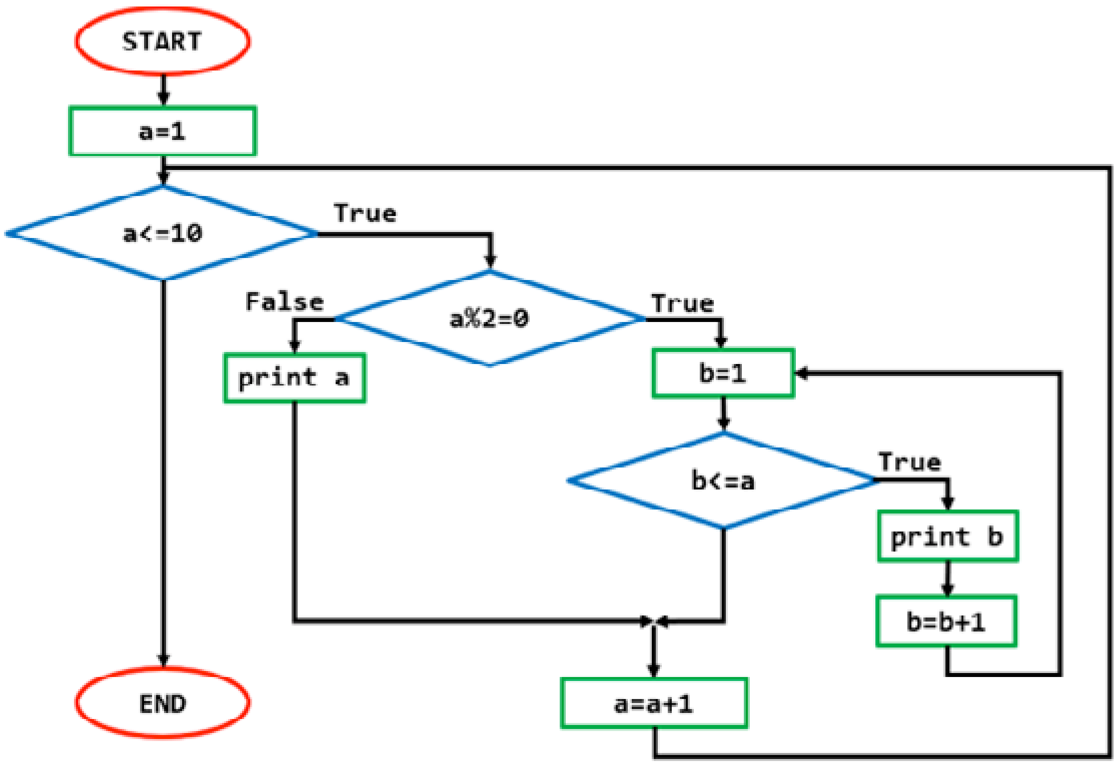
Exercise 7: Number evaluator



Exercise 8: Flowing while:+



Exercise 9: Flowing while:++



Lab 3: Working from solutions

In the previous Lab, we explored using flowcharts as a way to break down larger problems into smaller steps.

In this Lab, we shall invert this concept, by generating both the flowcharts and the code for several outputs which we have been given.

Solutions, steps, problems

It may be the case that, occasionally, during your time as a programmer, you may encounter a program whose code you do not have access to.

Alternatively, you may encounter a program which is difficult to read due to either complexity, bad programming, or unfamiliarity.

In these cases, it is important to understand how the program works by working backwards, instead of forwards. The program may be a solution to some problem, but you do not necessarily know what that problem is.

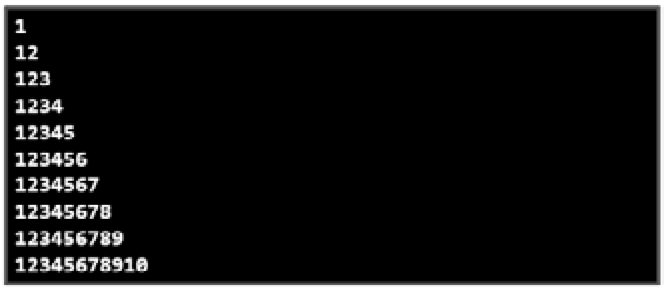
Outputs

An **output** is the end result of a program once it has been written.

While it is important to bear in mind that not all programs will necessarily generate outputs, each of the Exercises you will encounter in this Lab do.

Each Exercise consists of an output, which you must convert both into a flowchart and the code which produces it.

Let’s take a simple example of counting numbers from 1 to 10:



```
1
12
123
1234
12345
123456
1234567
12345678
123456789
12345678910
```

The output for this program is a series of numbers counting from 1 to 10.

From here, we know that we will need to generate two things:

1. A flowchart which counts from 1 to 10;
2. The code which gives this output.

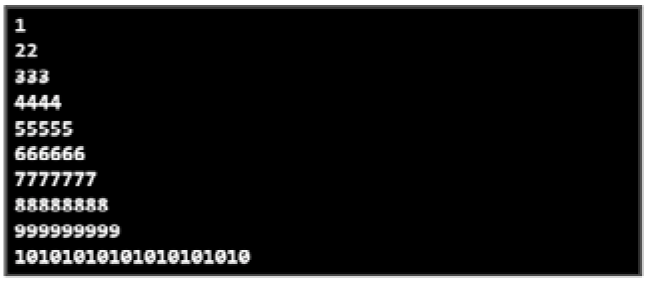
From this information, we can infer several things:

- We know that there will be at least one variable in this file, which is increased up until 10 before stopping.
- We can put this variable inside a while-loop and increment it by 1 until it reaches 10.
- As the output starts with 1, we know that the variable we increment through our while-loop will start at 1.

Exercise 1: 1-to-10 counter
Use the information we have about this program to create a flowchart for this output.

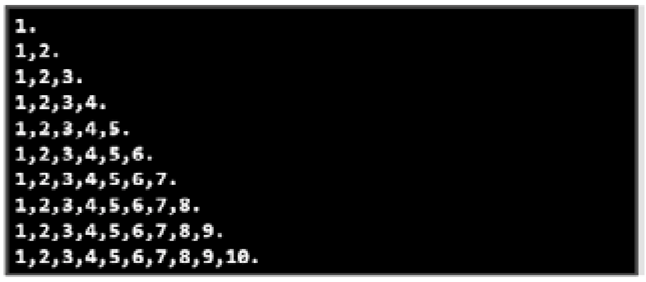
Once you have created your flowchart, write the code for it, as you have done for flowcharts that you have encountered in Lab 2.

Exercise 2: 1-to-10 duplicator



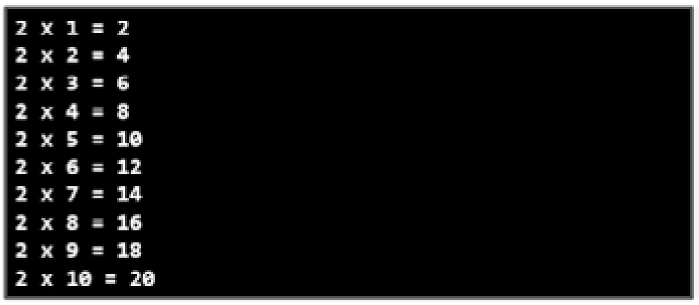
```
1
22
333
4444
55555
666666
7777777
88888888
999999999
10101010101010101010
```

Exercise 3: 1,-to-10. duplicator



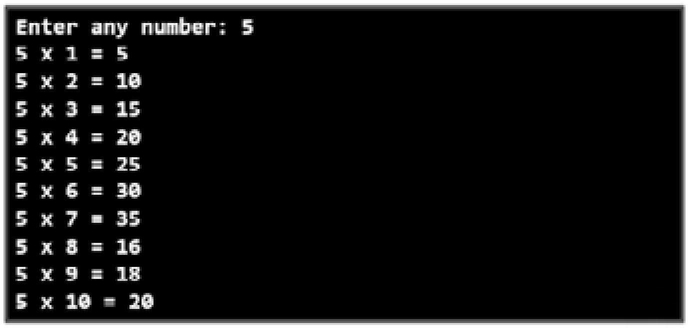
```
1.
1,2.
1,2,3.
1,2,3,4.
1,2,3,4,5.
1,2,3,4,5,6.
1,2,3,4,5,6,7.
1,2,3,4,5,6,7,8.
1,2,3,4,5,6,7,8,9.
1,2,3,4,5,6,7,8,9,10.
```

Exercise 4: 2 times tables up to 10



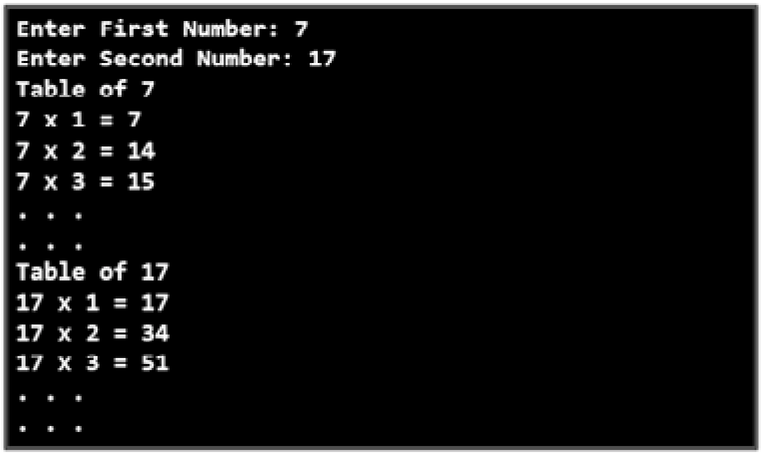
```
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
2 x 10 = 20
```

Exercise 5: input(x) times tables up to 10



```
Enter any number: 5
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 16
5 x 9 = 18
5 x 10 = 20
```

Exercise 6: input(x) times tables up to input(y)



```
Enter First Number: 7
Enter Second Number: 17
Table of 7
7 x 1 = 7
7 x 2 = 14
7 x 3 = 15
. . .
. . .
Table of 17
17 x 1 = 17
17 x 2 = 34
17 x 3 = 51
. . .
. . .
```

Summary

If you have attempted each Exercise, then everyone at Teach The Nation To Code would like to congratulate you on your efforts. Even if you did not get a chance to finish them all, or you did not think that you understood things, you will have still learned something useful today.

We hope that you have enjoyed TTNTC Recap as a companion to a Teach The Nation To Code Event, and that you might be able to attend an Event in the future.

If you feel confident in your abilities, and you are interested in learning programming as something more than just a hobby, the team at Teach The Nation To Code are part of the **QA Talent Academy** – you can learn more about starting a career in tech at qa.com/learners/become-a-digital-consultant/