



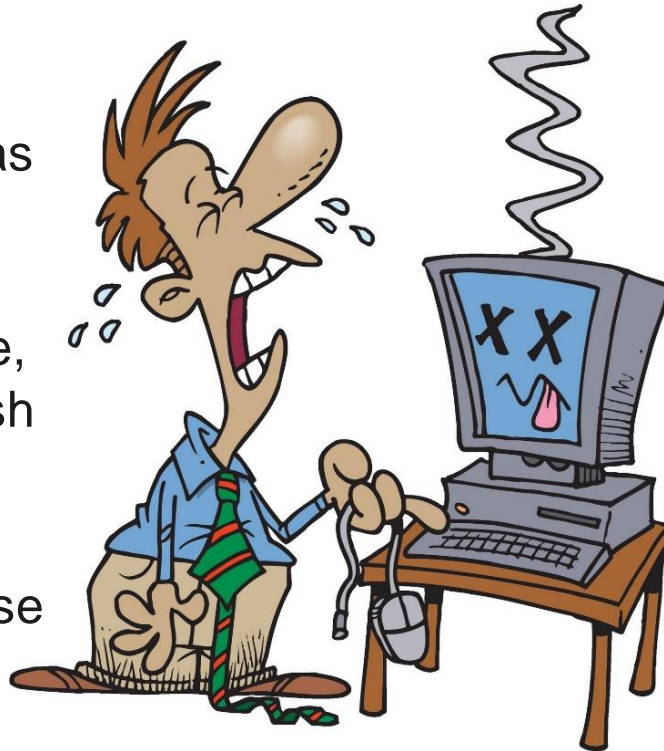
IFB104 — Building IT Systems

Topic 4 — How to Repeat Actions

School of Computer Science
Semester 1, 2025

Assessment 1A

- Assessment Task 1A is due at the today!
 - Deadline: Monday, March 17th, 11:59pm
 - You can submit as many drafts as you like—Do so *now*!
 - Your computer *will* die, your Internet connection *will* go offline, and the Canvas server *may* crash just before the deadline!
 - Make sure you successfully upload *several* drafts *before* these calamities happen
- Teaching staff will **not** answer your panicky emails late at night
 - Do **not** attempt to email Python files to teaching staff
 - Microsoft Outlook blocks Python attachments so we won't receive them!



Need help?

- *STIMulate* Peer Learning Facilitator consultation sessions
 - On-campus, Monday to Friday, 10am to 3pm
 - See the drop-in timetable at <https://stimulate.qut.edu.au/> (choose “IFB104” as the unit)
 - Online – submit a request for peer support via <https://qut.to/q826v>
- See the IFB104 Canvas site under *Unit Overview* | *Getting Help* for links




STIMulate

“Census date” is Friday

- This Friday is your last opportunity to withdraw from IFB104 without penalty
- Our job is to pass students, not fail them
- Assessment Task 1A is due in Week 4 so that you can decide whether or not to continue with IFB104 this semester
- The teaching unit gets more challenging as it goes on, so this is a good time to decide
- Full details about withdrawing from units can be found on the [Digital Workplace](#)

Fees - Census dates

Friday 16 August 2024

Census date is the date in each teaching period by which you need to have your enrolment requirements finalised. Failure to satisfy your enrolment requirements will result in cancellation of your enrolment. [More information about census dates](#) .

A timely plagiarism reminder

- Your submissions for Assessment Task 1 will be submitted to a software plagiarism analyser
- Submit your own work, not someone else's, even if it's incomplete
 - You'll get partial marks for an incomplete solution
- And *never* believe your best friend when they say they “just want to have a look” at your code “to see how you did it”

Finalisation of Misconduct - IFB104

Faculty of Science Appeals and Misconduct Committee

Sent: Tuesday, 15 November 2022 4:23 PM

ID	Last name	First name	Penalty
You don't want your name to appear here!			a 6 month exclusion and a fail recorded for the unit IFB104 Building IT Systems in semester 2, 2022

Assessment 1B

- Assessment Task 1A marking gets underway this week
 - But don't hold your breath – there are a lot of students enrolled this semester
 - We'll tell you when they're done
 - Aim to provide you feedback in 2 weeks at the latest, so you have time to take that onboard for assignment 2
- The next set of client's requirements, for Assessment Task 1B, appear on Canvas on Tuesday of week 4 (tomorrow)
 - The deadline is the end of Week 6 (Friday, April 4th)
 - Everything needed to complete the assignment is covered in Weeks 1 to 5
 - The submission will open later this week, as we are transitioning to Gradescope.

QUT Neuroinclusion Network

- NO DIAGNOSIS REQUIRED
 - Available to all QUT students, including those that identify as Neurodivergent
- <https://unihub.qut.edu.au/students/events?page=1&studentSiteId=1&text=neuroinclusion>

🔍 Search events

My event bookings My saved events

Keywords

neuroinclusion

Type of event

☐ Careers and professional development

☐ Orientation

☐ Resource booking

☐ Student life, social and wellbeing

☐ Study, research and learning

☐ Talks, seminars and lectures

Date start



Locations



Skills



Date added



Found 6 events

Starting soon ▼

 **NeuroInclusion Network - Social Study Hacks** 
Student life, social and wellbeing
20 MAR Thu 20 Mar 2025, 9:00 AM - 10:00 AM
📍 KG-R Block, QUT Kelvin Grove Campus
An opportunity to discuss study hacks and causally connect with other students who identify as Neurodivergent. Study hack topics will vary throughout the semester.

 **NeuroInclusion Network - Social Study Hacks** 
Student life, social and wellbeing
27 MAR Thu 27 Mar 2025, 2:00 PM - 3:00 PM
📍 Online
An opportunity to discuss study hacks and causally connect with other students who identify as Neurodivergent. Study hack topics will vary throughout the semester.

 **NeuroInclusion Network - Social Study Hacks** 
Student life, social and wellbeing
3 APR Thu 3 Apr 2025, 3:00 PM - 4:00 PM
📍 GP-V Block, QUT Gardens Point Campus
An opportunity to discuss study hacks and causally connect with other students who identify as Neurodivergent. Study hack topics will vary throughout the semester.

 **NeuroInclusion Network - Social Study Hacks** 
Student life, social and wellbeing
10 APR Thu 10 Apr 2025, 9:00 AM - 10:00 AM
📍 KG-R Block, QUT Kelvin Grove Campus
An opportunity to discuss study hacks and causally connect with other students who identify as Neurodivergent. Study hack topics will vary throughout the semester.

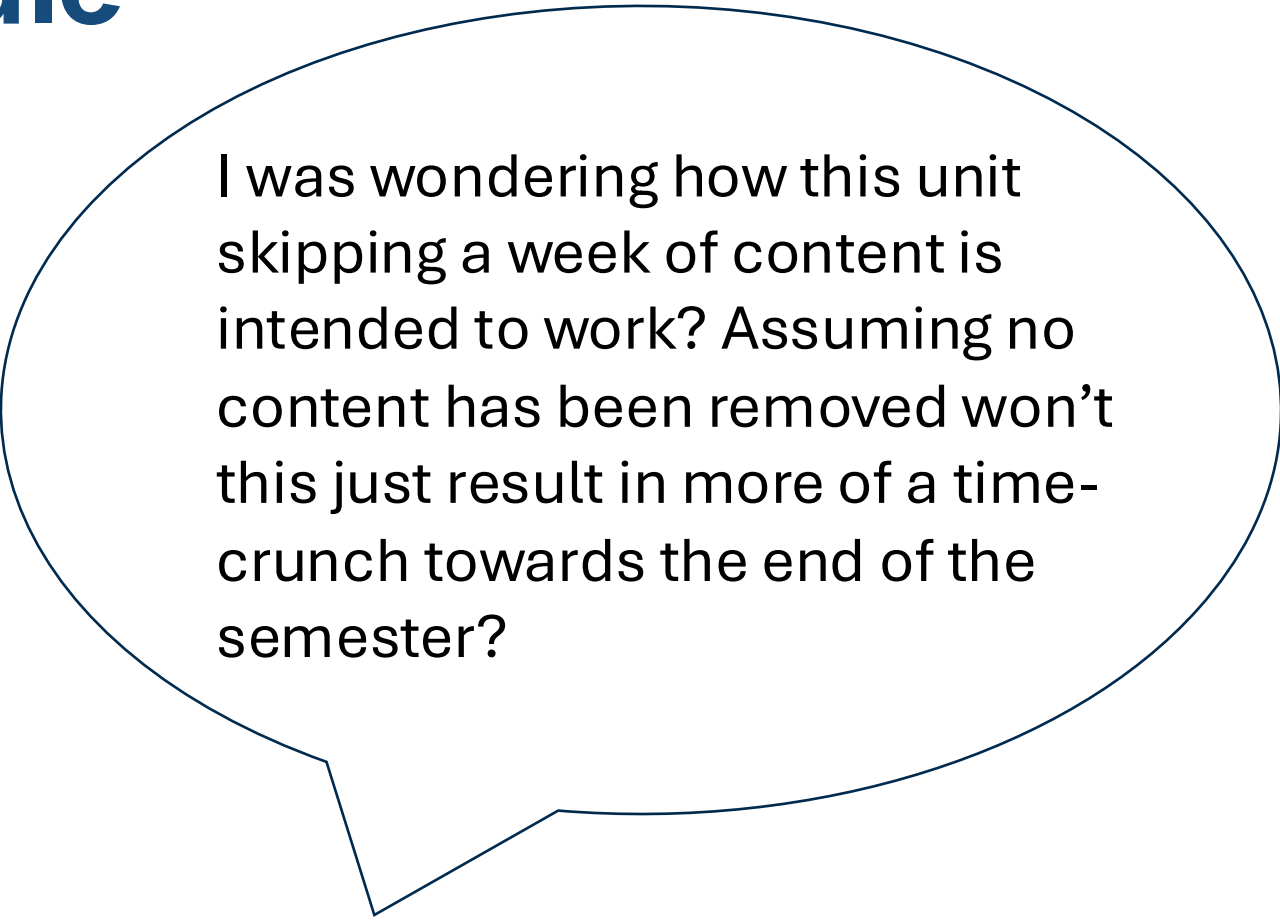
New syllabus schedule

- Change of due date for Assessment 1A
- Change of due date for Assessment 1B
- Shift new content from week 3 to week 4, from week 4 to week 5, from week 5 to week 6.
- Merge content from week 6 into week 7

3	10 March 25	No Lecture	Assessment 1A		
4	17 March 25	"How to Repeat Actions"	Lists and Loops	Assessment Task 1A due on Monday (7%)	Census Date
5	24 March 25	"How to Find Patterns"	Regular Expressions		
6	31 March 25	"How to Think Like a Computer"	Assessment 1B	Assessment Task 1B due on Friday (23%)	
7	7 April 25	"How to Interact With the User"	Modules and Interfaces		
	14 April 25	Mid Semester Break			
8	21 April 25	No Lecture	Interfaces (part 2)	Assessment Task 2A due on Friday (7%)	

New syllabus schedule

- Firstly, we have cancelled guest lectures in week 3 and 7 in order to accommodate the changes. Guest lectures are not assessable.
- The key difference is for week 7, which has 2 weeks merged.
 - Merging modules and interfaces makes sense, so it does not add a lot of content, just changes some of the exercises on modules
- Week 7 content is in fact spread over 2 weeks, with the break in between. There is less content to cover than in 2024, so we are confident that it will not add pressure.



I was wondering how this unit skipping a week of content is intended to work? Assuming no content has been removed won't this just result in more of a time-crunch towards the end of the semester?

This week

- In week 1 we saw that we can put several values of the same type in a single variable, a list. We also saw that strings are a little bit like lists of characters.
 - This week we look into lists more closely.
- In week 2 we saw how Boolean expressions and *decision structures* such as **if** statements allow computers to make choices about *which* action to perform
 - This week we introduce *repetition structures* as a way of controlling *how often* to perform an action
- In general there are three kinds of repetitive statements in modern computer languages:
 - Fixed iteration (**for**-each loops)
 - Condition-controlled iteration (**while** loops)
 - Recursive function calls



Part A — Lists: functions and uses

Lists

- A list is a collection of items in a specific order.
- Lists are similar to strings but they can be a sequence of any type of values, not just characters (e.g., integers, strings, objects).
- Lists use indexing to access elements.
 - Indexing starts at 0.

0	1	2	3	4	5
2	10	19	34	40	42

```
>>> lotto_numbers = [2, 10, 19, 34, 40, 42] # a list of numbers
>>> print('The third lotto number is', lotto_numbers[2])
The third lotto number is 19
>>> lotto_numbers[0] + lotto_numbers[-1]
44
>>> shopping_list = ['milk', 'bread'] # a list of strings
>>> print(shopping_list + ['coffee']) # we can add lists
['milk', 'bread', 'coffee']
>>> print('The second thing to buy is', shopping_list[1])
The second thing to buy is bread
>>>
```

0	1	2	3	4	5
milk	bread	coffee			

Functions for Lists

- Lists are similar to strings but they can be a sequence of any type of values, not just characters
- As with strings, many list operations return values
- But unlike strings, some list operations change the value of the list variable they are applied to
- See the *Python Standard Library* under [Built-in Types](#) | [Sequence Types](#) for many operations applicable to lists

```
>>> colours = ['red', 'green', 'blue'] # define a list
>>> colours.index('blue') # where is 'blue' in the list?
2
>>> colours.count('blue') # how often does it appear?
1
>>> colours.append('purple') # extend the list
>>> colours
['red', 'green', 'blue', 'purple']
>>> colours[1] = 'orange' # change an item in the list
>>> colours
['red', 'orange', 'blue', 'purple']
>>> colours.insert(2, 'white') # insert a new value at pos 2
>>> colours
['red', 'orange', 'white', 'blue', 'purple']
>>>
```

Some more commonly-used list operations

<code>L[n]</code>	returns the n^{th} item in list <code>L</code> , counting from zero
<code>L[m:n]</code>	returns the subsequence of list <code>L</code> from position <code>m</code> , inclusive, to position <code>n</code> , exclusive
<code>len(L)</code>	returns the length of list <code>L</code>
<code>max(L)</code>	returns the largest value in numeric list <code>L</code>
<code>sum(L)</code>	returns the sum of numbers in a numeric list <code>L</code>
<code>L.append(i)</code>	adds item <code>i</code> onto the end of list <code>L</code> (note that this operation changes <code>L</code> 'in place' rather than returning a value)
<code>L.sort()</code>	sorts the items in list <code>L</code> (again this changes variable <code>L</code> 's value in place)

Key programming concept: Two reasons for calling a function (or method)



- Most functions/methods accept some arguments and *return* a new value, which can be used in a larger expression, assigned to a variable, or printed to the screen
- However, some functions produce a *side-effect* on variables or the computing environment and return nothing
 - An attempt to access the value returned by a pure side-effecting function gets the special value `None`

```
>>> # import a constant from a module
>>> from math import pi
>>> # evaluate an expression and store the
>>> # value returned
>>> area = pi * (4 ** 2)
>>> area
50.26548245743669
>>>
>>> # define a list of values
>>> directions = ['up', 'down', 'left']
>>> # apply a function to the list that returns
>>> # a result
>>> len(directions)
3
>>> # apply a method to the list that has a
>>> # side-effect
>>> directions.remove('left')
>>> directions
['up', 'down']
```

Key programming concept: Mutable versus immutable variables



- String variables are immutable (unchangeable)
 - All string operations return new strings
 - To update a string variable we must assign a new value to it
- List variables are mutable (changeable)
 - Some list operations return new lists and leave the given list unchanged
 - Other list operations change the list 'in place' when the operation is applied
 - It's hard to remember which of these effects a particular string function has, so keep the *Python Standard Library* manual handy!

```
>>> name = 'Tweedledum' # create a string
>>> name[0:-2] + 'ee' # replace the last two letters
'Tweedledee'
>>> name # but the variable's value is unchanged
'Tweedledum'
>>> name = name[0:-2] + 'ee' # store the changed value
>>> name # now the variable has changed
'Tweedledee'

>>>
>>> letters = ['a', 'b', 'c'] # create a list
>>> letters + ['d'] # return a new list
['a', 'b', 'c', 'd']
>>> letters # but the variable is unchanged
['a', 'b', 'c']
>>> letters.append('d') # use an 'in place' function
>>> letters # now the variable's value has changed
['a', 'b', 'c', 'd']
>>>
```


Doing something for each item in a list

- We can do an action for however many values there are in a list
- For this reason we often refer to Python's `for` statements as “for each” statements

```
>>> # create a list of values
>>> winds = ['N', 'S', 'E', 'W']
>>> # do something for each of the values
>>> for wind in winds:
        print('Whoosh!')
```

```
Whoosh!
Whoosh!
Whoosh!
Whoosh!
```

Example: Taste Statistics

foods = [ ,  ,  , ]

for food in foods:

count = input ('how many of you like '+ food +'?')

Key programming concept: Accessing sequence values in for-each loops



- More generally, we can also refer to each of the values in the sequence via the variable named in the **for**-each statement
- This allows us to perform a *different action* at each iteration

```
>>> # Do a different action for each
>>> # letter in a character string
>>> for letter in 'WHAM':
        print('.o0', letter, '0o.')
```

For each value of this variable ...

taken from this sequence ...

```
for variable in sequence:
    statements
```

... do these statements, which may refer to the variable

```
.o0 W 0o.
.o0 H 0o.
.o0 A 0o.
.o0 M 0o.
```

Accessing sequence values in for-each loops

- The “target” variable in the **for**-each statement can be used in the loop body just like any other variable

```
>>> numbers = [45, 6, 23, 12, 9, 3, 25]
>>>
>>> # Print just the single-digit numbers
>>> # from the list above
>>> for number in numbers:
>>>     if number < 10:
>>>         print(number)
```

```
6
9
3
```

Lists of Lists

- A list of lists is a list where each element is itself a list.
- Useful for representing matrices, grids, or nested data structures.
- Use double indexing to access elements within nested lists.
 - The first index accesses the outer list, the second index accesses the inner list.

```
>>> # Some elements, grouped into families
>>> elements = [
    # Inert gases
    ['He', 'Ne', 'Ar', 'Kr', 'Xe'],
    # Halogens
    ['F', 'Cl', 'Br', 'I', 'At'],
    # Alkali earth metals
    ['Li', 'Na', 'K', 'Rb', 'Cs']]

>>> print(elements[0][1])
```

Ne

#Inert gases	#Halogens	#Alkali earth metals
He	F	Li
Ne	Cl	Na
Ar	Br	K
Kr	I	Rb
Xe	At	Cs

Key programming concept: Lists of lists



- For-each loops are the basic way of accessing each item in the list, one at a time
 - List items can be accessed either directly via their value or indirectly via their position
- When lists contain 'sublists' we can nest for-each loops accordingly

```
>>> # Some elements, grouped into families
>>> elements = [
    # Inert gases
    ['He', 'Ne', 'Ar', 'Kr', 'Xe'],
    # Halogens
    ['F', 'Cl', 'Br', 'I', 'At'],
    # Alkali earth metals
    ['Li', 'Na', 'K', 'Rb', 'Cs']]

>>>
>>> # Print elements with an 'a' in their symbol
>>> for family in elements:
    for symbol in family:
        if 'a' in symbol.lower():
            print(symbol)
```

Ar
At
Na

Part B — Doing (almost) the Same Thing Multiple Times

Doing the same thing many times

- The computational power of computers derives from their ability to rapidly repeat simple actions
 - We will explore repeated, or “iterative”, calculation in depth in a later lecture
- For now we just want to repeat the same action a fixed number of times, so that we can create some interesting images using Turtle
- The `for` statement repeats any actions indented below it for a specified `range` of iterations

```
>>> # print the same word to the screen 5 times
>>> for word in range(5):
    print('Hello')
```

```
Hello
Hello
Hello
Hello
Hello
```

Simple for-each loops

- We have seen how to do the same action several times using a **for**-each loop and the built-in **range** function
- In Python this is often combined with the **len** function to do a repeated action for however many items there are in a sequence

```
>>> # Do action once for each letter in word
>>> for letter_num in range(len(word)):
    print('Hi, there!')
```

```
Hi, there!
Hi, there!
Hi, there!
Hi, there!
Hi, there!
Hi, there!
```

More on the range function in for-each statements

- The **range** function can be used in **for**-each statements to access items in sequences via their position
- If given two arguments the **range** function produces indices from the first, inclusive, to the second, exclusive

```
>>> palindrome = 'Rats live on no evil star!'
>>>
>>> # Print just the second half
>>> midpoint = len(palindrome) // 2
>>> for index in range(midpoint, len(palindrome)):
        print(palindrome[index], end='')
```

no evil star!

Key programming concept: Exiting loops early



- For efficiency it is sometimes convenient to exit a loop before all iterations have been completed, e.g., when the answer has already been found before the end
- Two ways of doing this in Python are as follows:
 - A **return** statement executed from anywhere within a function, including from within a loop, will immediately exit the entire function
 - A **break** statement executed in a loop will cause the loop to end immediately
- Doing this makes the number of iterations performed by a 'definite' loop indefinite!

```
>>> palindrome = 'Rats live on no evil star!'
>>>
>>> # Print letters until a space is encountered
>>> for letter in palindrome:
        if letter == ' ':
            break
        else:
            print(letter)
```

```
R
a
t
s
>>>
```

Part C — Condition-controlled iteration

The need for condition-controlled iteration

- In all the examples so far, the length of a sequence tells us how many iterations will be performed
- In some situations we may not know how many iterations will be required:
 - Calculations that make progress towards a solution incrementally
 - Repeated actions controlled by external/user inputs
- In these cases we need a different kind of loop, the **while** loop, which keeps iterating as long as some condition is true

```
# Set up a variable to control the loop
keep_going = 'yes'

# Calculate and print a series of sales commissions
while keep_going == 'yes':

    # Get salesperson's sales and commission rate
    sales = input('Enter number of sales: ')
    comm_rate = input('Enter commission rate: ')

    # Calculate and display the commission
    commission = sales * comm_rate
    print('Your commission is', commission)

    # See if the user wants to continue
    keep_going = input('More (yes/no)? ')
```

Key programming concept: Condition-controlled iteration via `while` loops



- In Python condition-controlled iteration is achieved using **while** loops:

As long as this
Boolean condition is
true ...

while *condition:*
statements

... do these
statements,
which usually
must update
the variables in
the condition

```
# Set up a variable to control the loop
keep_going = 'yes'
```

```
# Calculate and print a series of sales commissions
while keep_going == 'yes':
```

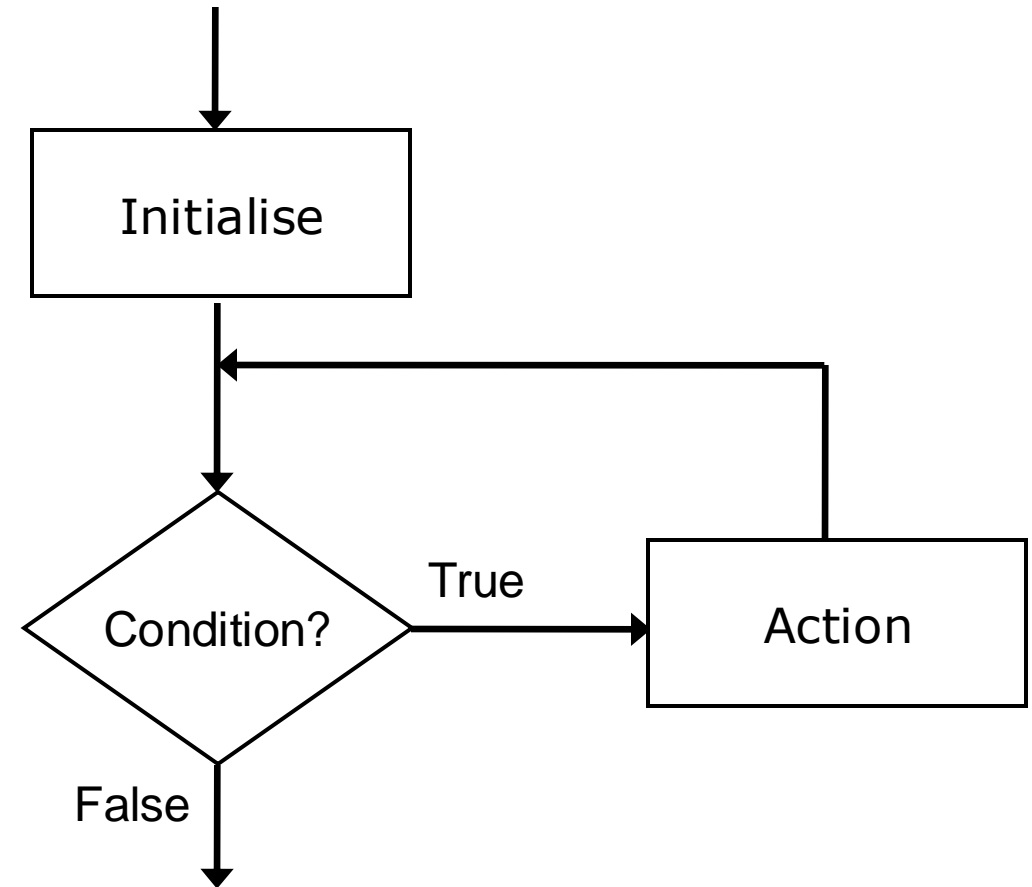
```
# Get salesperson's sales and commission rate
sales = input('Enter number of sales: ')
comm_rate = input('Enter commission rate: ')

# Calculate and display the commission
commission = sales * comm_rate
print('Your commission is', commission)
```

```
# See if the user wants to continue
keep_going = input('More (yes/no)? ')
```

Condition-controlled iteration via `while` loops

- The action in the loop body may be executed zero, one or many times, depending on the condition
- There is usually an initialisation action immediately before the `while` statement to establish the condition for the first time



Danger: Infinite loops!

- Condition-controlled iteration introduces the risk that the loop condition remains true forever
 - “Application is not responding”
- When designing a condition-controlled loop, think carefully about how the loop makes *progress* towards termination

```
# Initialise loop variable  
natural = 0
```

```
# This is an infinite loop! (Why?)  
while natural >= 0:  
    print(natural)  
    if natural < 10:  
        natural = natural + 1
```

Part D — Recursive functions

The recursion strategy

- Another way to perform actions repeatedly is to define *recursive* functions
- No new programming language statements are needed for recursive algorithms
 - Recursion simply involves *functions that call themselves*
 - Recursive solutions are usually more concise than iterative ones because they don't need variables to keep track of the computation's state
 - However recursive code is usually less efficient than iterative code
- Any problem we can solve with iteration can be solved recursively (and vice versa)

```
>>> def countdown(time):  
    if time == 0:  
        print('Lift off!')  
    else:  
        print(time)  
        countdown(time - 1)
```

```
>>> countdown(5)  
5  
4  
3  
2  
1  
Lift off!
```

Recursion in everyday life

- People often find recursion difficult to understand at first because it's an *unfamiliar* concept (*not* a hard one!) with few everyday analogues:

Onions:



Russian dolls:



'Infinite' mirrors:



Episodes of *Dr Who* in which the TARDIS materialises inside itself:



Danger: Infinite recursion!

- Recursive functions definitions are controlled by a Boolean condition, just like a **while** loop
- This means we can accidentally code *infinite* recursions (which consume not only processor time, but memory space!)
- To ensure that a recursive algorithm terminates, *every* recursive case must reduce the size of the remaining subproblem, bringing it closer to the base case

```
# Don't call this function!  
def never_ending_story():  
    print('Once upon a time ...')  
    never_ending_story()
```

Before next week ...

- Complete this week's workshop exercises and self-assessment test
- Start working on your solution to Assessment Task 1B
 - You still need to learn about regular expressions and testing next week in order to be able to complete the assignment, but you already have a lot to get started!