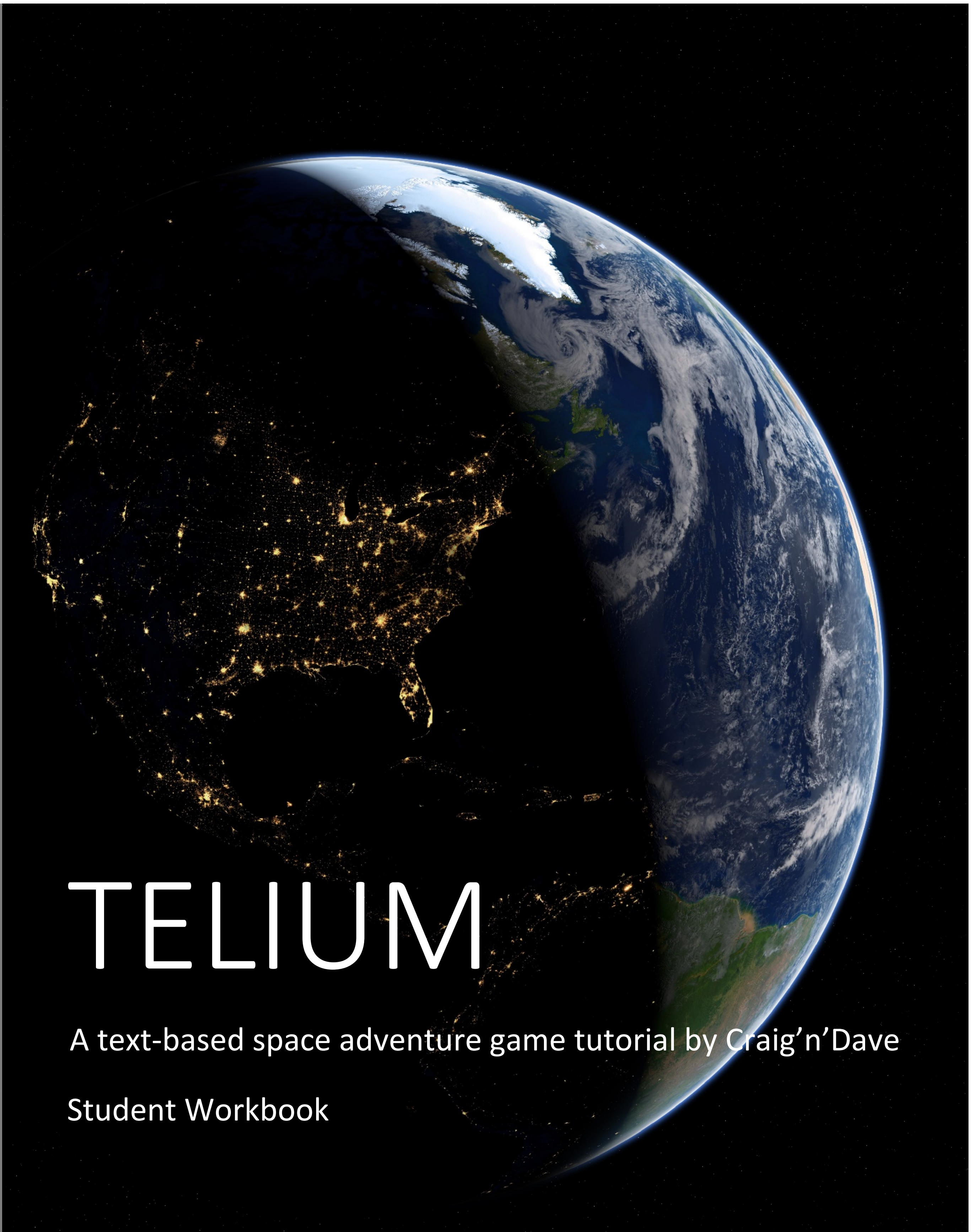


TELIUM

A text-based space adventure game tutorial by Craig'n'Dave

Student Workbook

Craig'n'Dave

Telium – the game

Telium [Tel-ee-um] is a text based adventure game based loosely on the films Life (2017) and Alien (1979). The game is coded in a number of stages. Each stage presents you with the base code to type in so you have a head-start and are not so daunted by the enormity of the production. The stage also includes additional challenges for you to extend the code further.

In the game, the player navigates around a space station called “The Charles Darwin” which is made up of a number of interconnecting modules. The object of the game is to find and trap a queen alien called “Telium” that is located somewhere in the station. Not wanting to get into a conflict with humans unless necessary, the queen will attempt to escape to adjacent modules when it is encountered. To win the game, the player must lock the appropriate module in the station so the queen cannot escape. Once trapped it can be killed with a flamethrower. A variety of additional objects are populated throughout the space station to enhance the adventure.

The lore

A remote probe on the surface of Mars has detected biological signatures of dormant, single celled primitive life. A sample of the Martian soil is returned to the space station orbiting the Earth for further analysis.

The sample of the orange coloured cells are examined and DNA analysis shows remarkable similarities to Dictyostelium discoideum, a species of soil-living amoeba here on Earth. Commonly referred to as slime mould, it transitions from a collection of unicellular amoebae into a multicellular organism and then into a fruiting body within its lifetime. Nicknamed, “Telium” due to its colour and cellular structure, the sample is incubated in the lab aboard the space station in conditions similar to when Mars was a warmer, wetter planet in its ancient past.

Remarkably, independent Telium cells start to slowly move and after a period of several days have joined together to form an organism resembling a slug. In further days the slug-like creature grows additional arms and begins to look like a large starfish. Each cell working with other cells to become a single organism. Intrigued, scientists continue to examine the creature that appears to be consuming bacteria from inside the incubation chamber, growing in size daily. Telium begins to show signs of advanced movement around the chamber and grows significantly in strength. Eventually becoming strong enough to break out of its chamber, suffocating the scientist examining it, the animal scuttles through the space station to an unknown location. The organism is not seen for several days, but tension between the astronauts escalates when the space station electronics begin to behave erratically, power starts draining and communication is lost with Earth. “We are on our own. Telium needs to be found and killed. There is no protocol for this and we cannot risk further loss of life. We must stick together and work it out”, the captain orders.

Mechanics of the game

The space station has a limited amount of power which reduces on each turn. This provides a timer for the game. Telium – the queen alien, must be killed before the power runs out.

The player is equipped with a flamethrower that requires fuel. This is used to kill aliens.

The player also has a portable computer called 'the scanner', that has limited functionality to interact with the space station. On each turn, the player can use their scanner to lock the doors in a module. Locking the doors prevents aliens from moving to that module. Due to the station malfunction, only one module can be locked at a time.

A player can move to an adjacent module.

Each module could contain:

- A ventilation shaft opening.
  In these modules the doors will lock on entry, forcing the player to move through the ventilation shaft. The dark passages lead to another random module. The player arrives in the new module and cannot return through the vent in the roof.
- The queen alien (Telium).
  When the player enters the same module as Telium it attempts to escape via random adjacent modules. It can take 1-3 moves. If it arrives in a module with a ventilation shaft it will travel down the shaft arriving in a random module.
  If Telium is unable to move due to the only adjacent module being locked the player wins.
- Worker aliens.
  Spawned asexually from the queen, the worker aliens gather bacteria for the queen to feed. They will attack the player if the player enters a module they are in. The player has the option to frighten the worker or attempt to kill it. This is done by using fuel from their flamethrower. The amount of fuel needed is not known by the player and will need to be deduced over several play attempts. The player will die if they do not frighten or kill the worker alien, losing the game.
- An information panel.
  This costs power to use, but scans the space station and reveals the location of the queen.
  The remaining power in the station is also be shown by the panel.

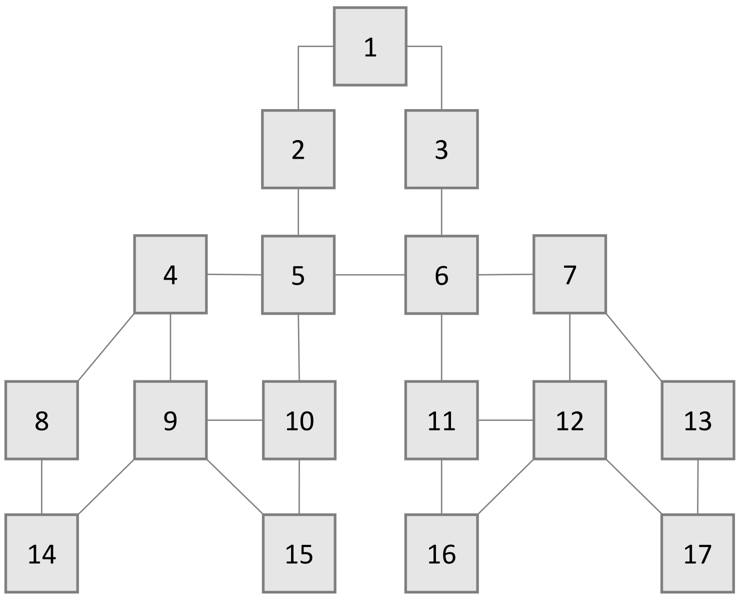
Your task

Type up the programs and then extend the code by completing challenges. There is plenty of scope to add to the game mechanics in many interesting ways once the program is complete.

You do not need to complete the challenges to progress to the next stage. The program will continue to work without them. You can also return to a previous challenge later if you want to.

Stage 1 – Creating the space station

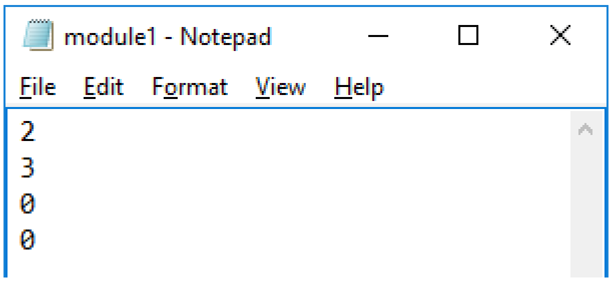
The station is a set of interconnecting modules. Each module can have up to 4 doors that lead to other modules. This can be illustrated as follows:



Here we see that module 1 has two doors, each connected to modules 2 and 3. Module 5 has four doors connected to modules 2, 4, 6 and 10 etc.

This space station is represented in 17 text files, named: module1.txt, module2.txt etc.

Each text file has 4 items of data stored inside it, representing the connected modules. A zero indicates the door is not connected.



```
2
3
0
0
```

This is the text file for module1.txt This module is connected to module 2 and 3 with no other connecting doors.

Note, each file must have 4 items of data otherwise the program will not work.

At least one module will need to have only 2 adjacent modules for the queen alien to be trapped. For game balancing, ideally this should be more than one, otherwise the game may be too difficult.

The program will work with any configuration as long as there are at least 10 modules:

- One for the player to spawn in (module 1).
- One for the queen alien to spawn in (random).
- Three for the information panels (random).
- Two for the ventilation shafts (random).
- Three for the worker aliens (random).

40 modules is probably a sensible maximum. The number of objects can be changed to affect the difficulty, with additional types of objects easily added. The number of objects cannot exceed the number of modules.

This is a good example of abstraction. How the space station looks on paper to how it is stored in files is very different.

Let's get started

- Create the 17 text files needed to represent a space station called "The Charles Darwin" using Notepad.
  Be very careful to ensure that:
  - each text file is named correctly – no spaces in the filename;
  - each text file has 4 items of data. Zero representing no door.

- Put the text files in a folder named `Charles_Darwin` – note the underscore in the folder name.

Learning Points

- Why is it important that each module has the same number of items of data even though not all the doors are connected? i.e. why bother with the zeros in the files?
- What are some of the advantages and disadvantages of storing the layout of the space station in text files?

Stage 2- Allowing the player to move around the space station

The program will read the data for the module the player is currently in to determine the valid moves the player can make. A zero in the text file representing no adjacent module for that door.

The player can be given a map of the space station when playing the game. It is only reasonable that an astronaut aboard the station would know their way around! However, in the panic and confusion they may not remember where objects are so these are hidden from the player.

Let's get started

- Create a new program in the folder above your "Charles_Darwin" folder to set up the basic navigation system for the game. Be very careful with the indentation. You will introduce logic or syntax errors if you make a mistake.

```
#Telium – The game

import random

#Global variables

num_modules = 17                #The number of modules in the space station
module = 1                      #The module of the space station we are in
last_module = 0                 #The last module we were in
possible_moves = []             #List of the possible moves we can make
alive = True                    #Whether the player is alive or dead
won = False                     #Whether the player has won
power = 100                     #The amount of power the space station has
fuel = 500                      #The amount of fuel the player has in the
flamethrower
locked = 0                      #The module that has been locked by the player
queen = 0                       #Location of the queen alien
vent_shafts = []                #Location of the ventilation shaft entrances
info_panels = []                #Location of the information panels
workers = []                    #Location of the worker aliens

#Procedure declarations

def load_module():
    global module, possible_moves
    possible_moves = get_modules_from(module)
    output_module()

def get_modules_from(module):
    moves = []
    text_file = open("Charles_Darwin\module" + str(module) + ".txt", "r")
    for counter in range(0,4):
        move_read = text_file.readline()
        move_read = int(move_read.strip())
        if move_read != 0:
            moves.append(move_read)
    text_file.close()
    return moves

def output_module():
    global module
    print()
    print("-----------------------------------------------------------------")
    print()
    print("You are in module",module)
    print()

def output_moves():
    global possible_moves
    print()
    print("From here you can move to modules: | ",end='')
    for move in possible_moves:
        print(move,'| ',end='')
    print()


def get_action():
    global module, last_module, possible_moves
    valid_action = False
    while valid_action == False:
        print("What do you want to do next ? (MOVE, SCANNER)")
        action = input(">")
        if action == "MOVE":
            move = int(input("Enter the module to move to: "))
            if move in possible_moves:
                valid_action = True
                last_module = module
                module = move
            else:
                print("The module must be connected to the current module.")

#Main program starts here

while alive and not won:
    load_module()
    if won == False and alive == True:
        output_moves()
        get_action()

if won == True:
    print("The queen is trapped and you burn it to death with your 
flamethrower.")
    print("Game over.  You win!")
if alive == False:
    print("The station has run out of power.  Unable to sustain life support, 
you die.")
```

Learning Points

- `num_modules` is a constant. What does that mean, and what are the advantages of using constants?
- The program has been written with a number of functions using the def command. Why?
- Why is the command `global` needed?
- Explain how has input been validated.

Challenges

- Comment the functions so each is explained.
- The power variable should decrement by one after each move to add a time limit to the game. If the power variable equals 0 the player dies due to a lack of life support.
- Improve the input sanitisation so the player does not need to enter MOVE. Lowercase and abbreviation to M should be accepted too.
- Create your own space station configuration by modifying the text files.
- Create a map of your station in a suitable application that a player can use to navigate in your game.
- Add an extra line to the module files that contains a name for the module and outputs this to the screen.
  e.g. "You are in module 1. This is the communication module where astronauts contact Earth."
- Add a title screen that allows the player to choose play, story, instructions or quit.
- Add an instruction screen that loads a page of instructions about how to play the game.
- Add extra text to the module files to output a description of the module to the player.
- Add an aspect of the story that the player can read.

Super Challenges

1. Create a simple text parser for the game so the player can enter MOVE and the module in the same line.
   E.g. an input of: MOVE 2 would move the player to module 2.

2. Further extend this to work with a range of input sanitisation options, e.g. M2, move2 has the same effect.
   Don't forget to update your player instructions screen with these new input options.

Stage 3- Spawning the NPCs

The following objects are in the game:

- The player. Attributes: current location, fuel (for the flamethrower) and station power.
- Queen alien (Telium). Attributes: current location.
- Ventillation shafts. Attributes: location.
- Information panels. Attributes: location.
- Worker aliens. Attributes: location.

Objects in the game are represented in lists. Each type of object has its own list of modules it exists in. E.g. workers = [4, 7, 8] would mean there was a worker alien in modules 4, 7 and 8.

Two objects cannot occupy the same module at the start of the game.

Let's get started

- Add the following procedure to your code:

```
def spawn_npcs():
    global num_modules, queen, vent_shafts, greedy_info_panels, workers
    module_set = []
    for counter in range(2,num_modules):
        module_set.append(counter)
    random.shuffle(module_set)
    i = 0
    queen = module_set[i]
    for counter in range(0,3):
        i=i+1
        vent_shafts.append(module_set[i])

    for counter in range(0,2):
        i=i+1
        info_panels.append(module_set[i])

    for counter in range(0,3):
        i=i+1
        workers.append(module_set[i])
```

- Under the comment `#Main program starts here`, add the following lines:

```
spawn_npcs()
print("Queen alien is located in module:",queen)
print("Ventilation shafts are located in modules:",vent_shafts)
print("Information panels are located in modules:",info_panels)
print("Worker aliens are located in modules:",workers)
```

Learning Points

- When you are developing the game, why is it necessary to output where the non-player characters (NPCs) have been placed when you want to keep this a secret from the player?
- How does the code work to ensure that two NPCs cannot occupy the same module?

Challenges

- Comment the code so each part is explained.
- When you enter a module containing an NPC, it should output, “There is a … in here”.

Super Challenges

- The game contains a queen alien, worker aliens, ventilation shafts and information panels. Can you think of additional game elements that could be spawned? Some ideas might include:

    - Additional fuel cells for the flamethrower. The game is quite difficult with a limited way of gaining fuel. Extra fuel canisters around the space station would help.

    - Power distributer. To be coded later: when found space station power is increased.

Stage 4 – Ventilation shafts

These teleport a player or queen alien to another random module. Their purpose is to add a small puzzle element to the game as some modules will be inaccessible by certain routes due to the placement of the ventilation shafts. A player has no option except arriving in a random module if they go into a room with a ventilation shaft. As a bonus they receive fuel for the flamethrower.

Let's get started

- Add the following procedure to your code:

```
def check_vent_shafts():
    global num_modules, module, vent_shafts, fuel
    if module in vent_shafts:
        print("There is a bank of fuel cells here.")
        print("You load one into your flamethrower.")
        fuel_gained = 50
        print("Fuel was",fuel,"now reading:",fuel+fuel_gained)
        fuel = fuel + fuel_gained
        print("The doors suddenly lock shut.")
        print("What is happening to the station?")
        print("Our only escape is to climb into the ventilation shaft.")
        print("We have no idea where we are going.")
        print("We follow the passages and find ourselves sliding down.")
        last_module = module
        module = random.randint(1,num_modules)
        load_module()
```

- In the main program section at the bottom of the code, underneath `load_module()` add the following line:

```
check_vent_shafts()
```

Learning Points

- What does the line: `if module in vent_shafts:` mean?
  i.e. What is the program actually checking?
- What does the line: `fuel = fuel + fuel_gained` do?
- What is the comma known as in programming in this line:
  `print("Fuel was",fuel,"now it is",fuel+fuel_gained+".")`

Challenges

- Comment the code so each part is explained.
- In the code, the player gains 50 fuel. Change this code so that the player gains either 20, 30, 40 or 50 fuel at random.

Super Challenges

- It is possible for the player to arrive back in the same module they have just escaped from, or for them to land in another module containing another ventilation shaft. Prevent this from happening.

Stage 5 – Locking a module

To win the game the player has to trap the queen alien (Telium). The queen will always try to escape if a player arrives in the same module. The queen will not go past the player to escape via the module the player has come from. Therefore, the queen can only be trapped in a module that has two exits.

When designing the space station, it is important that there are a number of places where this can happen. The player won't know how to trap the queen or indeed this is how the game is won until this is deduced through playing. This adds a puzzle element to the game. By locking certain modules and entering the same module as the queen, it can be coerced into a game winning position.

The game is won when the queen has no modules to escape to.

Locking a module costs power to force the player to think before taking this action since the game is lost if the station runs out of power. In order to make the game harder, only the exits in one module can be locked at any one time.

A module cannot be locked if it contains the queen alien.

The currently locked module is stored in a variable called `locked`.

Let's get started

- Add the following procedure to your code:
  Note the arrow indicates this is not a new line, but a continuation of the one before.

```
def lock():
    global num_modules, power, locked
    new_lock = int(input("Enter module to lock:"))
    if new_lock<0 or new_lock>num_modules:
        print("Invalid module.  Operation failed.")
    elif new_lock == queen:
        print("Operation failed.  Unable to lock module.")
    else:
        locked = new_lock
        print("Aliens cannot get into module",locked)
    power_used = 25 + 5*random.randint(0,5)
```

- In the get_action procedure, add the code in an appropriate place to handle the user choosing to lock a module when prompted for a move:

```
if action == "SCANNER":
    command = input("Scanner ready.  Enter command (LOCK):")
    if command == "LOCK":
        lock()
```

Learning Points

- In what ways is the validation insufficient when entering the module to lock?
- The space station should lose some power each time the player uses their scanner to lock a module. This does not happen, but why not?

Challenges

- Comment the code so each part is explained.
- Add a line of code to reduce the power from the station when a module is locked.
- Add a POWER command to the scanner that outputs the current power of the space station.
- The procedure does not prevent a player from locking a module that is already locked. Can you add that to the program?

Super Challenges

- The input is not fully exception handled. Can you prevent the program crashing if the user enters an invalid input for the module they want to lock?

- Create a simple text parser for the game so the player can enter LOCK and the module in the same line. E.g. an input of: LOCK 2 would lock module 2. This should work without having to input the word SCANNER first. This makes the game more user friendly for players who are familiar with the game.

- Further extend this to work with a range of input sanitisation options, e.g. L2, lock2 has the same effect.

- Add another command: LIFEFORMS that outputs how many aliens (queen and workers) are detected in the space station.

Stage 6 – The queen alien

When the player enters the same module occupied by the queen alien it will attempt to escape by making one, two or three moves to adjacent modules. The queen cannot go past the player to enter the module the player has just entered from. However, if it is possible for the queen to double-back behind the player, that is a valid move. If the queen enters a module containing a ventilation shaft it travels to a random module and stops moving irrespective of whether it had moves left to make.

The queen cannot enter a locked module. If the queen has nowhere to escape to the game is won, although the final boss battle is one of the challenges to be programmed.

This is the most complex section of code. It is a great opportunity to discuss problem decomposition, algorithmic thinking and testing.

Let's get started

- Add the following procedure to your code:
  Note the arrow indicates this is not a new line, but a continuation of the one before.

```
def move_queen():
    global num_modules, module, last_module, locked, queen, won, vent_shafts
    #If we are in the same module as the queen...
    if module == queen:
        print("There it is!  The queen alien is in this module...")
        #Decide how many moves the queen should take
        moves_to_make = random.randint(1,3)
        can_move_to_last_module = False
        while moves_to_make > 0:
            #Get the escapes the queen can make
            escapes = get_modules_from(queen)
            #Remove the current module as an escape
            if module in escapes:
                escapes.remove(module)
            #Allow queen to double back behind us from another module
            if last_module in escapes and can_move_to_last_module == False:
                escapes.remove(last_module)
            #Remove a module that is locked as an escape
            if locked in escapes:
                escapes.remove(locked)
            #If there is no escape then player has won...
            if len(escapes) == 0:
                won = True
                moves_to_make = 0
                print("...and the door is locked.  It's trapped.")
            #Otherwise move the queen to an adjacent module
            else:
                if moves_to_make == 1:
                    print("...and has escaped.")
                queen = random.choice(escapes)
                moves_to_make = moves_to_make - 1
                can_move_to_last_module = True
                #Handle the queen being in a module with a ventilation shaft
                while queen in vent_shafts:
```

Craig'n'Dave

```
                    if moves_to_make > 1:
                        print("...and has escaped.")
                    print("We can hear scuttling in the ventilation shafts.")
                    valid_move = False
                    #Queen cannot land in a module with another ventilation
shaft
                    while valid_move == False:
                        valid_move = True
                        queen = random.randint(1,num_modules)
                        if queen in vent_shafts:
                            valid_move = False
                    #Queen always stops moving after travelling through shaft
                    moves_to_make = 0
```

In the main program section, underneath `check_vent_shafts()` , add:

```
move_queen()
```

Learning Points

- What is the value of commenting large sections of code like this?
- What do we mean by the term, “problem decomposition”? How is this applied to this procedure to ensure it can be programmed successfully?

Challenges

- If the flamethrower contains less than 100 fuel, output a “low fuel” warning to the player on each turn.
- Draw up a test plan with sample data that could be used to test the procedure moving the queen alien.

Super Challenges

- The game immediately ends when the queen alien is trapped. Add a final boss battle where the player has to use all the fuel in their flamethrower in an attempt to kill it. The queen requires 100 to be killed as a minimum. If there is insufficient fuel the player is killed.

- In step 2 you may have considered some extra objects to spawn into the game. Program these now.

Stage 7 – Intuition

To make the game a little easier and more compelling, the player is given some prompts about what is in adjacent modules. This replaces their real-life senses. If there are worker aliens in the next module an astronaut says, “I can hear scuttling!” If there is a ventilation shaft in the next module the astronaut says, “I can feel cold air!”

Let’s get started

- Add the following procedure to your code:

```
def intuition():
    global possible_moves, workers, vent_shafts
    #Check what is in each of the possible moves
    for connected_module in possible_moves:
        if connected_module in workers:
            print("I can hear something scuttling!")
        if connected_module in vent_shafts:
            print("I can feel cold air!")
```

- In the main program section, underneath `if won == False and alive == True:` add:

```
intuition()
```

Learning Points

- Why are two `if` statements instead of `elif`?

Challenges

- Add extra code to check if the queen alien is in an adjacent module and output the message, “Listen! Did you hear that?”
- Add extra code to check if an information panel is in an adjacent module and output the message, “There is a panel near here. We could use it to find lifeforms.” Don’t forget the global list you will need to include too.

Super Challenges

- Add a new procedure to handle information panels. When a player enters a module with a panel they can choose to interact with it. If they do, the space station loses 50 power and the panel will output the location of the queen alien. The player cannot perform this action if power is less than 51.

- Messages will be repeated if there is more than one of the same NPC or object in an adjacent module, i.e. if there were two ventilation shafts in two adjacent modules the program would output: “I can feel cold air! I can feel cold air!” This is helpful to the player because they can deduce what is in adjacent modules, but the repeating messages feel odd. Change the program so that the message is not duplicated, a different message is output if there are 2 or 3 objects in adjacent modules. There is a good opportunity to use the scanner to output the message.

- Add a new function to the player scanner: SCAN. When a player selects to scan, they choose the module to scan and lose 25 power. The scanner tells the player what is in the module they have chosen to scan. A player can only scan an adjacent module. The player cannot perform this action if the power is less than 26.

Stage 8 – worker aliens

The worker aliens are placed in random locations at the beginning of the game. In the lore it is suggested they are produced asexually from the queen. Whilst it would be possible to program this, including movement from the queen's location, it is not necessary. Often in game design providing the player can believe the NPC actions are viable they don't need to absolutely follow the rules. For example, in racing games, often the NPC cars in front will slow down allowing the player to catch up. This makes the game more enjoyable and as long as the cars don't stop completely, the movement looks believable. In this game, since the player cannot know what is happening elsewhere in the space station it is entirely believable that the queen is spawning workers and they are moving.

The worker aliens provide a second mechanism by which the player can die. They either run out of moves (the space station power is exhausted) or they are killed by a worker alien. This adds challenge to the game.

When a worker alien is encountered the player has two choices: frighten the worker away or kill it. Frightening a worker just enables the player to continue to move, but the creature remains in the module for a future encounter. Killing it removes it from the game. Both actions require the player to use their flamethrower. Although in reality it would be far too dangerous to have a flamethrower on a space station, this is science fiction!

The player decides how much fuel to use. 30-80 is required to frighten a worker alien and 90-140 is required to kill a worker alien, although the player would not know this and will need to learn through experience. The random element here provides a mechanism for the player to panic and use either too little or usually, too much fuel.

If a player runs out of fuel during an encounter they die because the alien will attack and kill them. If the player uses too little fuel for their chosen action they must try again. It would be possible to build in a hit point/health mechanism into the game if able students wanted to do that.

In the base game, the player can never escape an encounter until they either frighten the worker alien or kill it. An extension is to be able to run back to the previous module.

Let’s get started

- Add the following procedure to your code:

```
def worker_aliens():
    global module, workers, fuel, alive
    #Output alien encountered
    if module in workers:
        print("Startled, a young alien scuttles across the floor.")
        print("It turns and leaps towards us.")
        #Get the player's action
        successful_attack = False
        while successful_attack == False:
            print("You can:")
            print()
            print("- Short blast your flamethrower to frighten it away.")
            print("- Long blast your flamethrower to try to kill it.")
            print()
            print("How will you react? (S, L)")
            action = 0
            while action not in ("S", "L"):
                action = input("Press the trigger: ")
            fuel_used = int(input("How much fuel will you use? ..."))
            fuel = fuel - fuel_used
            #Check if player has run out of fuel
            if fuel<=0:
                alive = False
                return
            #Work out how much fuel is needed
            if action == "S":
                fuel_needed = 30 + 10*random.randint(0,5)
            if action == "L":
                fuel_needed = 90 + 10*random.randint(0,5)
            #Try again if not enough fuel was used
            if fuel_used >= fuel_needed:
                successful_attack = True
            else:
                print("The alien squeals but is not dead. It’s angry.")
        #Successful action
        if action == "S":
            print("The alien scuttles away into the corner of the room.")
        if action == "L":
            print("The alien has been destroyed.")
            #Remove the worker from the module
            workers.remove(module)
        print()
```

- In the main program section, underneath `move_queen()` add:

```
worker_aliens()
```

Learning Points

- A game like this will need some "balancing". What does that mean in the context of computer games? What needs to be considered when balancing this game?

Challenges

- Add validation to prevent the program crashing if the user enters a string when prompted for how much fuel to use.
- Play the game as a real player to check the balance. Modify the variables as you need to in order to achieve a fun, but challenging game.

Super Challenges

- Add the option to RUN when a worker alien is encountered. If a player choses to do this, they return to the last module unless they arrived via a ventilation shaft in which case they retreat to a random connected module. There is a small possibility the player will be attacked and die if they choose this action.

- In the game there are a small group of 4 astronauts aboard the space station. If fuel runs out in an encounter, or if the player retreats but is killed the game ends. If there are characters remaining when one player dies they escape to a random connected module and get a new flamethrower with 250 fuel. When the last astronaut dies, or power is zero in the space station, the game is over.

- If an alien is frightened away but not killed, sometimes it should escape to another random module that does not contain another NPC.

Stage 9 – Your development

You now have the basic game working, but there are many ways you could make the game more enjoyable and interesting. Think about some possible extensions. How difficult would they be to program? What would you have to do to the code to implement it?

Discuss the easiest idea you have with your teacher and get their feedback.

Let's get started

- Have a go at programming your extensions. Don't forget to use functions and comments.