**Programming in Python with the Pi2Go Simulator: Part 3**

**Chapter 1: Lists**

**AIM:** After completing this chapter you should be able to use list data structures in your programs. In particular you should be able to access elements in a list, add items to a list, iterate over a list and join two lists together and extract a sub-list from a list.

**You Need:** To complete this chapter you should be familiar with the material in parts 1 and 2.

**If the simulator isn’t already running: Start the Simulator, Select the Initio Simulation and default\_world.xml, then start IDLE (open a *new IDLE window* if you have used IDLE to start the simulator).**

As the name suggests a Python list stores a list of things. For instance, we could have a list called actions of strings for each action the robot can take:

actions = ['forward', 'backward', 'left', 'right', 'stop']

We can access an element in a list by a number starting from zero. So, actions[0], is the string ‘forward’.

Consider the following program:

import simclient.simrobot as pi2go

import time

actions = ['forward', 'backward', 'left', 'right', 'stop']

initio.init()

element = 0

while element < 5:

 action = actions[element]

 if (action == 'forward'):

 pi2go.forward(10)

 time.sleep(3)

 elif (action == 'backward'):

 pi2go.reverse(10)

 time.sleep(3)

 elif (action == 'left'):

 pi2go.spinLeft(10)

 time.sleep(3)

 elif (action == 'right'):

 pi2go.spinRight(10)

 pi2go.sleep(3)

 else:

 pi2go.stop()

 element = element + 1

**Question 1:** What does it do?

Instead of using while and a counter to examine every element in the actions list, we can instead use for. for x in list: selects each element of list in turn and assigns it to the variable x. So, for instance to select each element of actions in turn and assign them to the variable action we could write:

for action in actions:

**Exercise 1:** Rewrite the above program so that it uses for instead of while and doesn’t use the variable element anywhere.

Lists can take elements of any type, for instance we can mix strings and numbers:

actions = ['forward', 3, 'backward', 2, 'left', 1, 'right', 5, 'stop']

**Exercise 2:** Write a program using the above list where the robot performs the action that comes first in the list for the number of seconds given second in the list, then the action that comes third for the number of seconds that comes fourth and so on.

**Useful List Functions:**

The following are useful functions on lists

*list.*append(*item*) adds *item* to the end of a list

>>> actions = ['forward', 'backward', 'left', 'right', 'stop']

>>> actions.append(‘new’)

>>> print(actions)

['forward', 'backward', 'left', 'right', 'stop', 'new']

len(*list)* len returns the length of *list.* So, for instance:

>>> actions = ['forward', 'backward', 'left', 'right', 'stop']

>>> num\_actions = len(actions)

>>> print(num\_actions)

5

*list.*remove(*item)* removes *item* from *list.*

>>> actions = ['forward', 'backward', 'left', 'right', 'stop']

>>> actions.remove(‘left’)

['forward', 'backward', 'right', ‘stop’]

*item* in *list* returns True if *item* is in *list* and False otherwise

>>> actions = ['forward', 'backward', 'left', 'right', 'stop']

>>> ‘backward’ in actions

True

>>> ‘other’ in actions

False

Use these functions to write the following programs:

**Exercise 3:** Write a program which starts the robot moving and takes readings from the ultrasonic sensor every 3 seconds. When it has 30 readings, it stops and prints out a list of all the readings (use append and len).

**Hint:** An empty list is written []

**Exercise 4:**  Write a program which tries each of the robot’s actions in a loop. If, after executing the action for 3 seconds, the robot is closer than 50 to an obstacle then the program removes that action and carries on trying the other actions, until there are no actions left (use remove).

**Testing Hint:** You may need to move obstacles around the robot to test that this is working.

**Exercise 5:** Create a program containing two lists, one of all actions and one of “permitted” actions. The program should try each of the robots actions in a loop. If the action is permitted then the robot should execute the action, otherwise it should print out a message that the action isn’t permitted.

**Lists within lists**

In Python a list may contain another list. For instance, you could create a list with three items containing an action the robot has taken and the values from the distance sensor before and after executing that action for three seconds. You could then store these lists all together in a larger list.

**Exercise 6:** Write a data logging program that loops through each of the robot’s actions. It takes a distance reading before each action and after each action and creates a list containing lists of three items (an action, a before sensor reading, and an after sensor reading). Once it has tried every action, it should print out the list.

**Chapter 2: Random**

**AIM:** After completing this chapter should be able to use Python’s random module to create random behaviour in a program.

**You Need:** To complete this chapter you should be familiar with the material in parts 1 & 2. You should also be able to use list data structures in Python (chapter 1)

**If the simulator isn’t already running: Start the Simulator, Select the Initio Simulation and default\_world.xml, then start IDLE (open a *new IDLE window* if you have used IDLE to start the simulator).**

Python has a special module random that can be used to create *pseudo-random* behaviour. It is difficult to create genuinely random behaviour using a computer, but we can generate behaviour that seems random using statistical methods. Pseudo-random number generators always produce the same sequence of results given the same starting *seed* value, but the sequence seems random to the observer. Normally programming languages, like Python, use the current time as the starting seed value so that the behaviour of the program changes depending upon when it is executed.

Like the time module and the simclient.simrobot module you can load the random module into a program using the command

import random

The random module contains the following functions that you can use:

* random.randint(a, b) – returns a random integer, N, such that a <= N <= b.
* random.choice(list) – returns a random element from the list.
* random.random() – returns a random float (decimal number) between 0 and 1 (including 0, but not including 1)
* random.seed(n) – sets the starting seed value for pseudo-random number generation to n.

**Exercise 1:** Write a program that will execute a random action from the list for a random amount of time between 1 and 5 seconds:

actions = ['forward', 'backward', 'left', 'right', 'stop']

**Hint:** You may want to use a program from Worksheet 19 as a starting point.

**Chapter 3: Dictionaries and Tuples**

**AIM:** After completing this chapter you should be able to use Python dictionary data structures in programs.

**You Need:** To complete the chapter you should be familiar with the material in parts 1 & 2. You should also be able to use Python list data structures (chapter 1) and theh random module (chapter 2).

**If the simulator isn’t already running: Start the Simulator, Select the Pi2go Simulation and square.xml, then start IDLE (open a *new IDLE window* if you have used IDLE to start the simulator).**

**Dictionaries** are a common data structure used in many programming languages. They let you *look up* a value based on some key word. So, for instance, you could associate each action the robot can take with the duration of the action:

action\_dictionary = {“forward”:3, “backward”:2, “left”:5, “right”:3, “stop”:2}

Each entry in the dictionary is a pair of a *key* and a *value* separated by a colon.

**Question 1:** In action\_dictionary what are the keys?

**Question 2:** What are the values?

You can access the value of an entry in a dictionary with the syntax *dictionary[key]*

Consider the following program.

import simclient.simrobot as pi2go

import time, random

pi2go.init()

actions = ['forward','backward','left','right','stop']

action\_dictionary = {"forward":3, "backward":2, "left":5, "right":3, "stop":2}

while True:

 action = random.choice(actions)

 if (action == "forward"):

 pi2go.forward(10)

 elif (action == "backward"):

 pi2go.reverse(10)

 elif (action == "left"):

 pi2go.spinLeft(10)

 elif (action == "right"):

 pi2go.spinRight(10)

 else:

 pi2go.stop()

 time.sleep(action\_dictionary[action])

**Question 3:** What does the program do?

You can loop through all the keys in a dictionary using

for *key* in *dictionary*:

This is similar to how you could loop through every element in a list.

**Exercise 1:** Modify the above program so that instead of selecting an action at random from the list actions it loops through each of the keys in action\_dictionary in turn and attempts that action for the set duration.

You can change the value associated with a key in a dictionary with the syntax

*dictionary[key]* = *value*

Consider the following program:

import simclient.simrobot as pi2go

import time, random

pi2go.init()

actions = ['forward','backward','left','right','stop']

dictionary = {"forward":0, "backward":0, "left":0, "right":0, "stop":0}

while True:

 action = random.choice(actions)

 if (action == "forward"):

 pi2go.forward(10)

 elif (action == "backward"):

 pi2go.reverse(10)

 elif (action == "left"):

 pi2go.spinLeft(10)

 elif (action == "right"):

 pi2go.spinRight(10)

 else:

 pi2go.stop()

 time.sleep(3)

 if (pi2go.irLeftLine()):

 dictionary[action] = dictionary[action] + 1

 if (pi2go.irRightLine()):

 dictionary[action] = dictionary[action] + 1

 if (dictionary[action] > 3):

 break

pi2go.stop()

print(dictionary)

**Question 4:** What does the program do?

Trying running the program when the robot starts on the black square and when the robot starts some way from the black square.

**Question 5:** What difference does it make where the robot starts?

**Exercise 2:** Modify your program so that it tries 20 actions at random adding one to their *score* if either of the line sensors detects a line at the end of the action. After 20 actions the program prints out the action with the highest score.

**More on Keys:** So far, we have been using strings as the keys for our dictionaries, but it is also possible to use numbers as keys. You can’t use lists as keys, but you can use a special kind of list called a *tuple*. A tuple is like a list except that you can’t change the values in a tuple.

You can create a tuple with the syntax

*tuple* = (*element1, element2, element3*)

and access elements of a tuple as

*tuple*[*element*]

Consider the following program:

import simclient.simrobot as pi2go

import time, random

pi2go.init()

actions = ['forward','backward','left','right','stop']

dictionary = {"forward":0, "backward":0, "left":0, "right":0, "stop":0}

rewards = {(1, 1):2, (1, 0):1, (0, 1):1, (0, 0):0}

while True:

 action = random.choice(actions)

 if (action == "forward"):

 pi2go.forward(10)

 elif (action == "backward"):

 pi2go.reverse(10)

 elif (action == "left"):

 pi2go.spinLeft(10)

 elif (action == "right"):

 pi2go.spinRight(10)

 else:

 pi2go.stop()

 time.sleep(3)

 dictionary[action] = dictionary[action] + rewards[(pi2go.irLeftLine(), pi2go.irRightLine())]

 if (dictionary[action] > 3):

 break

pi2go.stop()

print(dictionary)

**Question 6:** What does the rewards dictionary do?

**Exercise 3:** Modify the above program so that instead of the reward increasing when the line sensors detect a black surface, it increases when they detect a clear surface.

**Chatper 4: Data Structure Exercises**



**AIM:** This chapter provides additional programs using lists and dictionaries. It assumes familiarity with chapter 1-3.

You may want to use **square.xml** or **line\_following.xml** as the world for these exercises.

**Exercise 1**: Use a dictionary to create a set of rewards where the key is the value returned by the infra-red left line sensor and the reward is 1 if the sensor detects black and 0 otherwise. Write a program that will execute a random action and then print out the reward after the action has executed for 3 seconds.

**Exercise 2**: Create a dictionary of scores for each action. Modify your previous program so that the reward for an action gets added to its score. After 20 random actions the program prints out the total reward for each action.

**Exercise 3:** Adapt your program so that the reward dictionary keys are now lists. Each list is a pair of the value from the left line sensor and the right line sensor and the reward is 2 if both sensors detect black, 1 if 1 sensor detects black and 0 otherwise.

**Exercise 4:** Adapt your program again so that the total reward score for an action is divided by the number of times that action has been taken – to give an average reward for each action.

**Hint:** You will probably need to create another dictionary to keep track of how many times each action has been taken.

**Hint:** Depending upon your implementation you may also want to watch out for division by zero which will give an error.

**Chapter 5: Introduction to Machine Learning**

**AIM:** After completing this chapter you should be able to explain the basics of simple Machine Learning using rewards and program a reward table which matches states and actions to average rewards.

**You Need:** To complete this chapter you should be familiar with the material in parts 1 & 2. You should also be able to use Python Data Structures (chapters 1 & 3) and the random module (chapter 2).

**If the simulator isn’t already running: Start the Simulator, Select the Pi2Go Simulation and oval.xml, then start IDLE (open a *new IDLE window* if you have used IDLE to start the simulator).**

In the past few chapters you have been programming the virtual Pi2Go to take a random action and then you have been giving it a reward. This concept of actions and rewards is the basis of one form of machine learning. In this type of learning, the robot slowly learns over time which actions give it the best rewards and then uses this information to select actions.

Over the next few chapters we are going to attempt to program the virtual Pi2Go so that it learns how to drive around the edge of the oval in **oval.xml** world.

So far, we have only given rewards to actions based on how many of the line sensors can detect black beneath the sensor.

**Question 1:** Do you think just knowing which action gets the best reward in any situation is the best way to calculate what action should be taken? Explain your answer.

**States:** We are giving rewards based on the values returned by the two infra-red line sensors. At the end of WS21, we represented this as a tuple containing two values. We will refer to this tuple as the *state*. We have the following table of states and rewards using a dictionary:

|  |  |
| --- | --- |
| State | Reward |
| (1, 1) | 1 |
| (1, 0) | 2 |
| (0, 1) | 0 |
| (0, 0) | 1 |

We also want to associate our reward calculations with states. That is, we want to know what is the best action to take in some state. We therefore want to start out our program with some default reward dictionary like the following:

reward\_dictionary = {((1, 1), "forward"):0, ((1, 1), "backward"):0, ((1, 1), "left"):0, ((1, 1), "right"):0, ((1, 0), "forward"):0, ((1, 0), "backward"):0, ((1, 0), "left"):0, ((1, 0), "right"):0, ((0, 1), "forward"):0, ((0, 1), "backward"):0, ((0, 1), "left"):0, ((0, 1), "right"):0, ((0, 0), "forward"):0, ((0, 0), "backward"):0, ((0, 0), "left"):0, ((0, 0), "right"):0}

However, it is quite tedious to create a dictionary like this by hand, so we will use a function to do it.

**Exercise 1:** Write a function that will return the action-reward dictionary above by using nested loops to generate each entry.

**Hint:** The function range(a, b) will return a list of all the numbers between a and b including a and not including b. So range(0, 2) returns the list [0, 1].

**Exercise 2:** Modify the function so it will take a list of actions and the default starting reward as input and return the dictionary that pairs each those actions with a state pair.

**Exercise 3:** Write a program that takes actions at random. It records the state of the system before the action was taken and after the action (actions should run for 3 seconds) was taken. It then calculates the total reward earned over time for each action in each state. After 50 actions it stops and prints out the average reward for each action in each state.

**Watch Out:** Watch out for division by zero when calculating averages. Your program should print out a “never attempt” message if it has never attempted some action in some state.

**Question 2:** Have you got any state-action pairs that have not been attempted? Y/N

Try re-running your program where you move the robot so it starts a long way outside the oval, and so it start in the black near the edge of the oval.

**Question 3:** Does the initial placement of the robot make a difference to the number of state-action pairs that have not been attempted?

**Chapter 6: Exploration versus Exploitation in Machine Learning**

**AIM:** After completing this chapter you should be able to explain how to use a learning rate to gradually move a machine learning program from exploring behaviours to selecting behaviours that are likely to yield a reward.

**You Need:** To complete this chapter you should be familiar with the material in parts 1 & 2. You should be able to use Python data structures (chapters 1 & 3), the random module (chapter 2) and done the exercises in chapter 5.

**If the simulator isn’t already running: Start the Simulator, Select the Pi2Go Simulation and oval.xml, then start IDLE (open a *new IDLE window* if you have used IDLE to start the simulator).**

So far you have written programs to that attempt to learn the best action to take in any situation by taking an action at random. However, when writing such a program, you want to gradually move from a phase in which the program *explores* different actions to see which one works best in a state and towards a phase in which it *exploits* it learning by selecting the best action. We can do this by having a number, let us call it *epsilon*, that gradually reduces as the program progresses.

Each time the program must select an action it generates a random number. If the number is higher than epsilon, then the program selects the action which currently has the highest expected reward. If the number is lower than epsilon, then the program selects an action at random.

We gradually reduce epsilon (for instance we can start with epsilon at 1 and reduce it by 0.05 each time the program earns a reward of 2).

**Exercise 1:** Introduce epsilon into your program from chapter 5. This program should attempt to learn how to move around the oval by gradually moving to a state where it has successfully learned what to do.

**Hints:**

* You will need to write an algorithm for selecting the action with the highest reward that you can use when the random number generated is higher than epsilon.
* When testing your program, you may need to move the robot back to the oval if it moves too far away from it.
* This program takes a while to run so you may find it useful to use print statements to show when epsilon reduces and whether an action is being selected at random or because it is the current best choice.

**Chapter 7: Learning Rate**

**AIM:** After completing this chapter you should be able to explain how to use a learning rate to gradually move a machine learning program from exploring behaviours to selecting behaviours that are likely to yield a reward.

**You Need:** To complete this chapter you should be familiar with the material in parts 1 & 2. You should be able to use Python data structures (chapters 1 & 3), the random module (chapter 2) and you should have completed the exercises in chapter 6.

**If the simulator isn’t already running: Start the Simulator, Select the Pi2Go Simulation and oval.xml, then start IDLE (open a *new IDLE window* if you have used IDLE to start the simulator).**

Your machine learning program currently calculates the average reward of an action over all attempts. However, more commonly, machine learning program use a number, called the *learning rate* to change the stored reward after each attempt. In this case we calculate the *difference* between the currently stored reward and the new reward (by subtracting the current reward from the new reward) and the modify the reward by this difference multiplied by the learning rate.

So, for instance, if the currently stored reward is 1.5, the new reward is 1.4 and the learning rate is 0.2 then we first calculate the difference between the two rewards

Difference = 1.4 – 1.5 = -0.1

Multiply this by the learning rate

Modified difference = -0.1 x 0.2 = -0.02

And then add this to the current reward

New reward = 1.5 + ( - 0.02) = 1.48

We can represent this as an equation where *R* is the current reward, *r* is the new reward and ** is the learning rate then:

R = R + (r – R) x 

**Exercise 1:** Replace the use of average reward in your program from chapter 6 with a learning rate of 0.5. Change the default value of all rewards from 0 to 1, since when working with a learning rate it is good start with a reward value somewhere in the middle of the range

**Chapter 8: Exercises with Machine Learning**

**Exercise 1:** Write a Machine Learning algorithm to get the virtual Pi2Go to avoid obstacles. It should choose between the actions forward, right and left, and use the input from the three infra-red obstacle sensors as its state.

**Exercise 2:** Write a Machine Learning algorithm to get the virtual Pi2Go to follow a wall by keeping the wall always to one side of it.

**Exercise 3**: Write a Machine Learning algorithm to get the virtual Pi2Go to turn towards a bright light.

**Hints:** You may want to use the following function to calculate the state. This returns a tuple where the first number determines whether it is brighter on the front left or the front right, the second whether it is brighter on the back left or back right and the third number whether it is brighter at the front or the back.

def get\_state():

 fl = pi2go.getLightFL()

 fr = pi2go.getLightFR()

 bl = pi2go.getLightBL()

 br = pi2go.getLightBR()

 comp\_f = 0

 if (fl > fr):

 comp\_f = 1

 elif (fr > fl):

 comp\_f = -1

 comp\_b = 0

 if (bl > br):

 comp\_b = 1

 elif (br > bl):

 comp\_b = -1

 comp = 0

 if (fl + fr > bl + br):

 comp = 1

 elif (bl + br > fl + fr):

 comp = -1

 return (comp\_f, comp\_b, comp)

You may also want to use the following function to calculate the reward rather than using a dictionary. This function increases the reward as the values of the two front light sensors get closer to each other:

def calculate\_reward():

 fl = pi2go.getLightFL()

 fr = pi2go.getLightFR()

 bl = pi2go.getLightBL()

 br = pi2go.getLightBR()

 if (bl + br > fl + fr):

 return 0

 f\_diff = fl - fr

 f\_diff\_mod = f\_diff/1023

 if (f\_diff\_mod > 0):

 return (1 - f\_diff\_mod)

 else:

 return (1 + f\_diff\_mod)

**Chapter 9: File Input/Output**

**AIM:** After completing this chapter you should be able to read information into a program from a file, and write data from a program to a file.

**You Need:** To complete this chapter you should be familiar with the material in parts 1 & 2. You should be able to use Python data structures (chapters 1 & 3), the random module (chapter 2) and you should have completed the exercises in chapter 7.

**If the simulator isn’t already running: Start the Simulator, Select the Pi2Go Simulation and oval.xml, then start IDLE (open a *new IDLE window* if you have used IDLE to start the simulator).**

Once your machine learning program has learned which actions to take in some state it would be nice if it these could be saved and reused.

In order to do this, you need to be able to read and write data from files.

**Writing to Files**

In order to write to a file, you must first open the file, you can do this with the method:

>>> f = open(‘datafile’, ‘w’)

In the above call datafile is the name of the file and w is the *mode.* In this case w means the file is being opened in *write only mode*. **Open** returns a *file object* – so f is a file object for datafile.

Once a file is open for writing you can write to it using the method write(*string*)

Once you have finished writing to a file you should close it using f.close()

Consider the following program:

dictionary = {((1, 1), "forward"):0.5, ((1, 1), "backward"):0.5, ((1, 1), "left"):1, ((1, 1), "right"):0, ((1, 0), "forward"):1, ((1, 0), "backward"):0.5, ((1, 0), "left"):0, ((1, 0), "right"):0, ((0, 1), "forward"):0.5, ((0, 1), "backward"):0.5, ((0, 1), "left"):0, ((0, 1), "right"):1, ((0, 0), "forward"):0.5, ((0, 0), "backward"):0.5, ((0, 0), "left"):0, ((0, 0), "right"):1}

f = open(‘policy\_file’, 'w')

 for key in dictionary:

 f.write(str(key))

 f.write(“::”)

 f.write(str(dictionary[key]))

 f.write('\n')

 f.close()

**Note:** \n is the “new line” symbol, so when you print or write \n the next print or write command appears on a new line.

**Question 1:** What does the program do?

**Exercise 1:** Adapt your machine learning program from chapter 7 so that it writes the policy it has learned to a file.

**More on File Modes:** There are four file modes:

* w – Write Only
* r – Read Only
* a – Append new data to the end of the file
* r+ -- Read and Write

**Reading from Files:** There are several ways to read information from a file, f:

* You can read all the information from at once using the function f.read()
* You can read the data one line at a time using the function f.readline()
* You can loop through the lines of the file using: for line in f – each time the program goes around the loop, the variable line will take the value of the next line in the file.

**Exercise 2:** Write a program that will read in the data from policy\_file that was created in the first program in this worksheet:

**Turning Strings in to Values:** If we are successfully going to use the information stored in our policy\_file then we are going to have extract the information about state, action and reward from our strings and turn them back into numbers and tuples. There are a number of functions that we can use for this.

* **Treating strings like lists:** We can treat strings like lists and use the [n] syntax to extract the *nth* character from the string for instance,

Try typing the following:

>>> s = “(0, 1)”

>>> s[0]

>>> s[1]

**Question 2:** What happens and why?

* **Extracting ranges:** We can extract a range of letters from a string using the syntax: [a:b] where a is the first letter to be extracted and b is the last.

Try typing the following:

>>> s = “(0, 1)”

>>> s[1:4]

**Question 3:** What happens and why?

* **Splitting strings.** Lastly, we can *split strings* by nominated a character and turning the string into an array of strings which appear between each appearance of that character.
* Sometimes when using split you need to use a backslash in front of the character if it is a quotation mark or similar that might indicate the end of the string.

Try typing the following:

>>> s = “(0, 1)”

>>> s.split(‘,’)

**Question 4:** What happens and why?

Consider the following program:

f = open('policy\_file', 'r')

for line in f:

 [key, value] = line.split('::')

 [a, b, c] = key.split(',')

 irR = a[2]

 irL = b[1]

 [x, action, y] = c.split('\'')

 print(irR)

 print(irL)

 print(action)

 print(value)

**Question 5:** What does the program do?

**Exercise 3:** Write a program that will read in the policy from policy\_file and then use it to control the Pi2Go to follow the black oval until the switch is pressed.

**Hint:** You will need to use int and float to convert strings to integers and floats.

**Chapter 10: Handling Exceptions**

**AIM:** After completing this chapter you should be able to explain how to catch exceptions in Python programs to prevent them from crashing the program.

**You Need:** To complete this chapter you should be familiar with the material in parts 1 & 2. You need to know how to use Python list data structures (chapter 1) and file I/O (chapter 9).

**For this worksheet you do not need the PiRover Simulator. Just open an IDLE window.**

Sometimes something goes wrong during program execution causing the program to crash.

Type the following at the command line.

>>> f = open('no\_file', 'r')

**Question 1:** What happens?

.

If this line was included in a program, then the program would crash.

**Exercise 1:** Write a program that is supposed to read in a file and then print its contents to the screen but give the name of a file that doesn’t exist.

**Question 2:** Run the program. What happens?

We can prevent this kind of thing happening by using **try … except.**

Consider the following program:

try:

 f = open('no\_file', 'r')

 for line in f:

 print(line)

except:

 print("error")

In this program if the code within the try: block fails with an error then the code in the except: block is executed. This is particularly useful when handling user input and input from files:

**Exercise 2:** Write a program that asks the user to input a number and then prints out that number times 10. If the user enters something that is not a number, then the program should note the fact and ask them to try again.



 University of Liverpool, 2020

This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-nc-sa/4.0/)