**Programming in Python with the Initio Simulator: Part 2**

**Chapter 1: Variables and Comparisons**



**AIM:** After completing this chapter you should be able to use variables to store values in Python programs and be able to compare numbers.

**You Need:** To complete this chapter you should be familiar with the material in Part 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).**

When we program, we frequently want to store a value and reuse it later. For instance, we might want to take two readings from one of the robot’s sensors, one after the other, and compare them.

Create a file containing the following program:

import simclient.simrobot as initio, time

initio.init()

reading1 = initio.getDistance()

time.sleep(10)

reading2 = initio.getDistance()

if (reading1 < reading2):

print("Object is moving away")

elif (reading2 < reading1):

print("Object is moving closer")

else:

print("Object is not moving")

In this program reading1 and reading2 are *variables.* We use them to store the values of two readings from the distance sensor taken 10 seconds apart and then compare them using < (less than).

**Remember:** elif, this means *else if*. So, the program has three options 1) if reading1 is less than reading2, 2) else if reading2 is less than reading1 3) else - this last option happens if both readings are the same.

**Question 1:** What do you think this program will do?

**Question 2:** How can you test if the program works?

Test the program now. Does it behave as you expect?

**Exercise 1:**  Modify the program to make the robot move towards an object that is moving away from it.

**More on Comparisons:** As well as checking if one number is less than another we might want to check if they are equal, or not equal. We can use the following **comparison operators** in Python:

|  |  |
| --- | --- |
| a < b | a is strictly less than b |
| a > b | a is strictly greater than b |
| a <= b | a is less than or equal to b |
| a >= b | a is greater than or equal to b |
| a == b | a equals b |
| a != b | a does not equal b |

**Exercise 2:** Write a program that uses == and != to decide whether an object is moving or not. It should print out Object Moving! if the object is moving and Object Stopped! if the object has stopped

**Exercise 3:**  Write a program to make the robot “chase” an object by adding a while loop. So long as an object is moving away from it the robot will keep moving towards the object, but the robot will stay still if the object is stationary and reverse if the object is moving towards it. You can also add in use of the left or right distance sensor to stop the program when something gets close enough to that sensor to trigger it.

**Chapter 2: Command Line Input/Output**

**AIM:** After completing this chapter you should be able to get input from a user to control your program and compare strings.

**You Need:** To complete this chapter you should be familiar with the material in Part 1 and know how to use variables in Python programs (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).**

It is often useful to get input from a user about how they want a program to run. To do this you can use the Python command input.

Create a file containing the following program and run it.

import simclient.simrobot as initio

initio.init()

input(['Press any key to get a distance reading'])

print(initio.getDistance())

**Question 1:** What happens when you press any key after this program runs?

Obviously, it would be good to access the input from the user. This is easy since input returns the value that the user enters. The value that the user enters is a *string.*

Try running the following program:

import simclient.simrobot as initio

initio.init()

name = input(['Please enter your name'])

print("Hello")

print(name)

print("The Distance reading is")

print(initio.getDistance())

**Question 2:** What does this print out?

It would be nicer if we could join together the Hello and the name. If something appears in a Python program between " symbols then it is also a string. So, "Hello" is also a string in the program. You can join two strings together using +. So, for instance, you could change

print("Hello")

print(name)

to

print("Hello " + name)

**Question 3:** Modify the program and run it again.

Did this work? Y/N

**Question 4:** Now try changing

print("The Distance reading is")

print(initio.getDistance())

to

print("The Distance reading is " + initio.getDistance())

Did this work? Y/N

You probably got an error that looked something like:

TypeError: can only concatenate str (not "float") to str

This is because initio.getDistance() returns a real number (called a *float*) not a string. If we want to join a string and a number together we have to convert the number into a string. This is can be done using the Python command str.

Try changing the line to

print("The Distance reading is " + str(initio.getDistance()))

Just like you can use str to change a number to a string, you can use int to turn a string to a whole number (an integer) and float to turn a string to a real number.

**Exercise 1:** Use str to write a program which asks the user how many distance readings they would like to get and then prints out that number of readings.

Another big use for input is to ask the user if they would like something done. For instance, you might want the program only to print out the distance reading if the user responds Y to a question. To do this you can use == to compare to strings, for instance with the line

if (input == ‘Y’):

**Exercise 2:** Use string comparison to write a program that asks the user if they want a distance reading. If they do it prints out the reading, if not then the program exits.

**Chapter 3: Basic Data Types**

**AIM:** After completing this chapter you should be able to describe the basic data types from Python, and some useful functions for manipulating them.

**You Need:** To complete this chapter you should be familiar with the material in Part 1. You should be able to use variables, numbers (chapter 1) and strings (chapter 2) in Python programs.

**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).**

We’ve met several different *data types* already in these worksheets:

**Integers** Whole numbers such as 1, 2, 340 and so on

**Floats** Real numbers with decimal points such as 1.5, 2.73

**Strings** words such as ‘forward’, ‘Y’, “hello world!”

**Booleans** True and False

It is also possible to represent complex numbers in Python, but we do not consider those here.

**Doing Mathematics: Integers and Floats**

You can do normal mathematics with integers and floats using the following symbols:

|  |  |
| --- | --- |
| + | Plus |
| - | Minus |
| \* | Times |
| / | Divide |

Consider the following program:

import simclient.simrobot as initio

import time

initio.init()

count = 0

total\_distance = 0

while (count < 20):

total\_distance = total\_distance + initio.getDistance()

time.sleep(3)

count = count + 1

print("The Average Distance is: " + str(total\_distance/count))

**Question 1:** What is printed out when you run it? (It takes 1 minute to run)

**Question 2:** Now add a block to the work and run the program again moving the block backward and forward in front of the virtual Initio? What is printed out?

**Exercise 1:** Modify the program so that it prints out the total distance measured over 10 measurements.

**Exercise 2:** Write a program that will take readings from the distance sensor until a total distance of over 1000 has been measured and then prints out the average distance per reading.

**Working with Strings**

In Python strings are surrounded by either single or double quotes. But what do we do if we want to include a quote symbol in a string?

We use an *escape* which is the backslash symbol \. So, for instance we can use \” to include a double quote symbol in a string:

Try running the following program:

import simclient.simrobot as initio

name = input(['Please enter your name'])

print("Hello \"" + name + "\"")

**Question 3:** What happens?

You can also use the escape symbol to create special characters like new line symbols and tab symbols.

|  |  |
| --- | --- |
| \n | New line |
| \t | Tab |

**Exercise 3:** Write a program that uses new line and tab to ask someone their first name and then their surname then prints Hello followed by a tab then their first name and then prints their second name on a new line.

**Casting**

Remember that you can use the function **str** to turn a number into a string for printing. This is called *casting* from a number to a string. In the same way you can cast from a string to a number using **int** and to a float using **float**

**Exercise 4:** Write a program that asks the user to enter an integer. It then multiples that integer by 100 and prints out the result.

**Chapter 4: Variables, Numbers, Booleans and Strings**



**AIM:** This chapter provides additional programming exercises using variables, numbers and strings. It assumes familiarity with chapters 1-3.

**Exercise 1**: Write a program that will take two readings from the ultrasonic sensor one second apart. If they are not equal it will print out a message containing the two readings and stating that they are not the same. This message should all be printed on one line.

**Exercise 2**:  Write a program will take a reading from the left and the right infrared sensors. If the readings are the same it will reverse for 10 seconds, otherwise it will turn away from whichever sensor detects something.

**Exercise 3**: Write a program which asks the user to write either forward or backward and then asks them to enter a number under 10. The robot should then move either forwards or backwards for the relevant number of seconds. If any of the inputs are incorrect the robot does nothing.

**Exercise 4**: Write a program which asks the user to write either forward or backward and then asks them to enter a number under 10. The robot should then move either forwards or backwards for the relevant number of seconds. If the user enters an incorrect input then the robot should explain the problem and prompt for the input again.

**Exercise 5:** Write a program which loops asking the user to input F, B, L, R (for forward, backward, left, right) and gets the robot to move in whichever direction they last indicated. As it moves it should print out the value from the ultrasonic sensor every 5 seconds.

**Exercise 6:** Write a program which loops asking the user to input F, B, L, R (for forward, backward, left, right) and gets the robot to move in whichever direction they last indicated. It then continues to ask this question (with an additional option of S for stop) and changes direction or stops as indicated by the user.

**Exercise 7:** Write a program which asks the user to enter a time in seconds. The robot should then move forward for that number of seconds. While doing so it should take an ultrasonic sensor reading every second and at the end of the program print out the average value of the ultrasonic sensor while it was moving.

**Chapter 5: Debugging**

**AIM:** After completing this chapter you should be able to identify problems in programs based on error messages and use the IDLE debugger.

**You Need:** To complete this chapter you should be familiar with the material in Part 1. You should be able to use variables, numbers (chapter 1) and strings (chapter 2) in Python programs.

**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).**

You have written a number of programs by now and so are probably already familiar with some of the error messages you might get.

Consider the following program. It contains an error:

import simclient.simrobot as initio

initio.init()

direction = input(["Which way would you like the robot to move? (F, B, L, R)"])

while(direction != "S"):

if (direction == "F"):

initio.forward()

elif (direction == "B"):

initio.reverse(10)

elif (direction == "L"):

initio.spinLeft(10)

elif (direction == "R"):

initio.spinRight(10)

direction = input(["Which way would you like the robot to move next? (F, B, L, R, S)"])

initio.stop()

If you run this program and type F as input you should see the following error message:

Traceback (most recent call last):

File "/Users/louisedennis/PiRovers/pirover\_simulator/examples/tmp2.py", line 12, in <module>

initio.forward()

TypeError: forward() missing 1 required positional argument: 'speed'

Read the error message carefully.

**Question 1:** On what line has the error occurred?

**Question 2:** What does the error message say is missing from forward() ?

An *argument* is something that goes in the brackets. You can fix this program by putting something in the brackets for forward().

**Exercise 1:** Correct the program and check that it works.

**Debuggers**

Sometimes problems in programs can not be found simply by looking for a syntax error. Many programmers use print statements to help them understand what their program is doing and what the values of variables may be but there is a better way to do this and that is by using a *debugger.* There are a lot of different debuggers and each one works differently key features however are

* **Code stepping** a debugger will let you execute a program one command at a time. This will let you see what is happening at each stage of the program, which branches of if statements are being used and similar things.
* **Breakpoints** in long programs simply stepping through every instruction can get tiresome, particularly if you know that the error doesn’t occur until later in the program. Debuggers will let you set a *breakpoint* in your code and then execute the program until the breakpoint is reach at which point you can start stepping through the code.
* **Inspecting Variables** Debuggers let you see what the values of variables are which can help you figure out what is going wrong.

We are going to look at the debugger that comes with IDLE.

To start the debugger, you should click on the **Debug** menu in IDLE’s Python Shell window and select **Debugger.** A window a bit like the following should appear and the words [DEBUG ON] will appear in the Shell window.

A screenshot of a cell phone

Description automatically generated

The Debugger Window

Run the Python Program you have been working on. You will notice that it doesn’t run, but a lot of information appears in the Debugger window such as \_\_file\_\_ showing the file name. The five buttons **Go, Step, Over, Out** and **Quit** are available.

**Question 3:** Click on **Go**. What happens?

Stop the program either by selecting S when prompted by the program.

Now run the program again and this time click **Over** instead. **Over** lets you step through each line in the program (we will discuss the use of **Step**) in a later tutorial.

In the top part of the debugger window you can see the line of code that the debugger is currently at – for instance

* ‘\_\_main\_\_’.<module>(), line 9: while(direction != “S”):

Is at line 9 of the program – the command while(direction != “S”):

The bottom part of the debugger window contains values of variables like direction.

Step through the code entering all the different letters F, B, L, R, S as prompted by the program.

**Question 4:** How many times do you have to click **Over**?

**Question 5:** Run the program again and click **Over** a couple of times and then click **Go.** What happens?

Lastly we will look at how to set a **breakpoint.** In the program right click on the line

elif (direction == "L"):

and select **Set Breakpoint**. The line should go yellow.

**Question 6:** Now run the program and click **Go.** Enter R when prompted by the program. What happens?

**Question 7:** What line have you stopped at?

**Question 8:** What is the value of direction?

**Chapter 6: Debugging Exercises**

**Exercise 1:** Consider the following program. It is supposed to run for 1 minute, taking a distance reading every three seconds and then print out the average distance. It contains two errors use the error messages and IDLE debugger to find them.

import simclient.simrobot as initio

import time

initio.init()

count = 0

total\_distance = 0

while (count < 20):

total\_distance = total\_distance + initio.getDistance()

time.sleep(3)

count = count + 1

average = total\_distance \* count

print("The Average Distance is: " + average)

**Exercise 2:** Consider the following program. What is wrong with it? Use the IDLE debugger to find the cause of the problem and propose a fix.

import simclient.simrobot as initio

import time

initio.init()

count = 0

total\_distance = 0

while (total\_distance < 20):

time.sleep(3)

count = count + 1

print("Done")

.

**Chapter 7: Functions**

**AIM:** After completing this chapter should be able to describe what a function and a function argument is in programming and use functions (with and without arguments) and functions which return values in your Python Programs.

**You Need:** To complete this chapter you should be familiar with the material in part 1. You should be able to use variables, numbers (chapter 1) and strings (chapter 2) in Python programs.

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

Sometimes we have parts of a program that perform the same sequence of commands.

Consider the following program that drives the robot forward for 5 seconds and shakes its head and then drives it backwards for five seconds and shakes its head.

import simclient.simrobot as initio

import time

initio.init()

initio.forward(10)

time.sleep(5)

initio.stop()

initio.setServo(1, 20)

time.sleep(5)

initio.setServo(1, -20)

time.sleep(5)

initio.setServo(1, 0)

initio.reverse(10)

time.sleep(5)

initio.stop()

initio.setServo(1, 20)

time.sleep(5)

initio.setServo(1, -20)

time.sleep(5)

initio.setServo(1, 0)

It would be good to separate out the code for shaking the head, so we didn’t have to type it all twice. We can do this using a *function*. Functions in programs are like mini-programs that can be executed to perform some small task. You can *call* functions from within larger programs. All the commands you have been using with the virtual Initio such as **forward, getDistance, init**  and so on are functions.

To create a function you use the keyword def followed by the name of the function and then open and close brackets and a colon. You then write the code for the function indented (like with **if**  and **while**) on the lines below. So, we can write a “shake head” function as follows:

def shake\_head():

initio.setServo(1, 20)

time.sleep(5)

initio.setServo(1, -20)

time.sleep(5)

initio.setServo(1, 0)

The program becomes:

import simclient.simrobot as initio

import time

def shake\_head():

initio.setServo(1, 20)

time.sleep(5)

initio.setServo(1, -20)

time.sleep(5)

initio.setServo(1, 0)

initio.init()

initio.forward(10)

time.sleep(5)

initio.stop()

shake\_head()

initio.reverse(10)

time.sleep(5)

initio.stop()

shake\_head()

**Exercise 1:** Write a program that will move forward until it detects an obstacle (something closer than 50cm). At that point it stops and shakes its head. Then it turns until it no longer detects an obstacle (at which point it shakes its head again). Then it moves forward until it detects another obstacle, stops and shakes its head. Use the shake\_head() function above for all the shaking.

**Functions with Arguments**

We don’t have to just use functions when we want to run the *exact same* piece of code again. We can use them if we want to run similar, but slightly different pieces of code. To do this we provide the functions with *arguments.* These appear in between the brackets after the function name. The arguments are names of variables that can then be used inside the function.

Consider the following function:

def turn(side):

if (side == 'left'):

initio.spinLeft(10)

else:

initio.spinRight(10)

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

**Question 2:** What does the following program do?

import simclient.simrobot as initio

import time

def turn(side):

if (side == 'left'):

initio.spinLeft(10)

else:

initio.spinRight(10)

initio.init()

turn('right')

time.sleep(10)

initio.stop()

**Exercise 2:** Write a function that takes either **‘left’** or **‘right’** as an argument and, if it detects an obstacle on that side, turns away from it until it no longer detects an obstacle. Show the use of this function in a program.

**Functions that Calculate Values**

Suppose we want a function that will calculate a value for a variable? We can use the return keyword to return the value

Consider the following function:

def obstacle(side):

if (side == 'left'):

return initio.irLeft()

else:

return initio.irRight()

**Question 3:** What does it do?

**Exercise 3:** Write a program that uses the obstacle function to print out the value from each of the two infra-red distance sensors

**Exercise 4:** Write a function, opposite(side), that returns ‘left’ if its argument is ‘right’ and vice versa.

**Exercise 5:** Using your opposite(side) function, the obstacle(side) function and the turn(side) function write a function turn\_until(side) that turns away from an obstacle until it is no longer picked up by the infra-red sensor on that side. Illustrate its use in a program.

**Chapter 8: Functions Exercises**

**AIM:** This chapter provides additional programs using functions.

**Exercise 1:** Create a function when\_obstacle\_close() which waits until and obstacle is closer than 15cm from the distance sensor before continuing – and sleeps for 3 seconds to give time for the obstacle to be moved away. Illustrate its use in a program which starts the robot moving once and obstacle appears briefly in front of it and stops the robot when obstacle appears again.

You can imagine this allowing you to stop and start the program by waving a hand in front of the distance sensor. It is easy to “wave” an obstacle in front of the robot sensors using the mouse and blocks from the edit window.

**Exercise 2:** Create a function forward\_for(t) where t is *time in seconds*. Which moves the robot forward for t seconds. Illustrate its use in a program which prompts the user to enter a time in seconds and then moves the robot forward for that number of seconds.

**Exercise 3:** Create a function average\_distance() which returns the average value from the ultrasonic sensor taken over ten readings taken 1 second apart. Illustrate its use in a program that drives the robot forwards, takes ten distance readings, stops and prints out the average distance.

**Exercise 4:** Create a function follow\_line() that moves forward if neither line sensor detects anything and turns left if the left line sensor detects something and right if the right line sensor detects something. Test this out in **line\_following.xml** world by placing the robot so that its line sensors are either side of the black line.

**Exercise 5:** Create a function avoid\_obstacle() that implements obstacle avoidance behaviour and a function follow\_line(). Illustrate their use in a program that prompts the use to select whether the robot should either avoid obstacles or follow a line or exit the program. If an obstacle is moved closer than 10cm then the program should prompt for a new instruction.

**Exercise 6:** Write a function furthest() which returns either ‘left’ or ‘right’ depending upon which side an obstacle is further away (you will need to use the servo to pan the distance sensor to decide this). Illustrate its use with a program that will turn towards the away from obstacles using the turn(side) function from the worksheet.

**Chapter 9: Modules**

**AIM:** After completing this chapter you should be able to explain what a namespace is in programming and use create and use modules in Python.

**You Need:** To complete this chapter you should be familiar with the material I part 1. You should be able to use variables (chapter 1), data types (chapter 3) and Functions (chapter 7) in Python programs.

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

You have already met the **import** keyword at the start of your programs which you have been using to import useful functions into your programs. These functions are stored in *modules* such as the **time** module.

You can create your own modules by storing all your functions in a file.

Create a file called turning.py this should contain the functions from WS16: turn(side) and obstacle(side)

You will need to import simclient.simrobot as initio and time at the start of this module as usual.

Type the following at the command line:

>>> import turning

>>> turning.initio.init()

>>> turning.turn('left')

>>> turning.initio.stop()

**Question 1:** What happens?

**Notice** that you have to use the file name turning at the start of all the functions you want to use in the file.

You have created a *module* which has the *namespace* turning.

If you want to change the namespace then you can use the import … as … syntax, like you have been doing for import simclient.simrobot as initio

**Important:** Type

>>> turning.initio.cleanup()

Before you import the module again for the next question.

**Question 2:** What sequence of commands would you need to type to import turning using the namespace my\_turning and then turn the virtual Initio to the right?

**Reloading a Module**

Although you can give a module a new namespace by reloading it. The Python command line only loads modules once. So, if you edit your module and try importing again it won’t use your changes.

**Exercise 1:** Edit your module so that it prints out a message when it executes the **turn** function. Then import it again and try executing turn.

What happens and why?

In order to properly reload the module you need to use something called **importlib.** Type the following:

>>> import importlib

>>> importlib.reload(turning)

Now try executing your new version of the turn function.

**Question 3:** What happens?

**Running code when you import a module.**

Suppose you wanted to initialise the Initio whenever you loaded the module so you didn’t have to type turning.initio.init() at the Python command line? This is easy, you just include the command in the module – just as in a normal program file.

**Exercise 2:** Adapt your module so that that it initialises the Initio when it is imported.

**Chapter 10: Wall Following**

**AIM:** After completing this chapter you should be able to integrate your new knowledge of Python programming in order program a simple wall following algorithm in Python.

**You Need:** To complete this chapter you should be familiar with the material in part 1. You should be able to use variables, numbers (chapter 1), data types (chapter 3) and functions (chapter 7) in Python programs.

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

**Challenge:** Create a program, using functions for turning left or right which will drive around the outside wall of the “house” in the **house.xml** world in one direction until it enters the house and then will reverse out and drive back around the house in the other direction until it enters again.

You can go ahead and attempt to write the program now – or you can follow the suggested steps below.

**Step 1:** Write a follow\_wall(side) function which moves forward if it detects an obstacle on side and turns towards side when it does not detect an obstacle to that side. You may want to get it to move forward a little and then turn when it doesn’t detect an obstacle on the desired side (this will help prevent the robot getting “stuck” turning towards and then away from a wall.

**Step 2:** Modify your function so that if it detects an obstacle in the centre (you will need to use the distance sensor for this and decide how close something need to be to count as an obstacle) it turns away from side until it doesn’t detect an obstacle in the centre *or* an obstacle on side

**Step 3:**  Write a program that will follow a wall on one side until it detects a black “floor” beneath it. Then will reverse backwards for a few seconds, turn back the way it came for a few seconds and then follow the wall on the other side.

**Step 4:** Finally add a function drive\_to\_wall() which will drive forward until the robot detects an obstacle and, at that point, will start following the wall around the house.

.

**Exercise 1:** Create a module from your wall following function and use it to create a set of functions that are useful for getting the Initio to explore the **maze1 maze2** and **maze3** worlds.



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/).