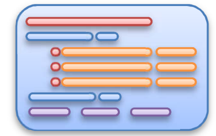
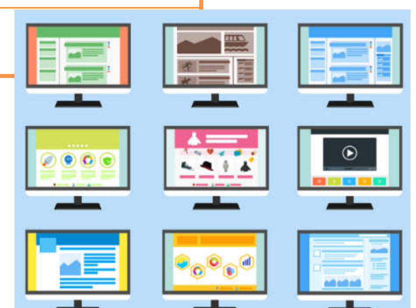
Programming in Python

Orb education

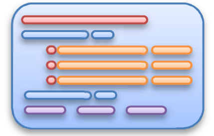


Do This	Section
☐	1. Computational Thinking
☐	2. Control Flow in Algorithms
☐	3. Flowcharts and Pseudocode
☐	4. Iteration in Scratch
☐	5. Robot Maze in Scratch
☐	6. Programming Languages
☐	7. Getting Started with Python
☐	8. Data Types and Variables
☐	9. Commenting
☐	10. Errors and Tracing
☐	11. Strings
☐	12. Maths Functions
☐	13. Lists
☐	14. Sorting Lists
☐	15. Searching Lists
☐	16. Arrays
☐	17. Procedures
☐	18. Creating Procedures
☐	19. Scope
☐	20. OOPL
☐	21. Creating a Game



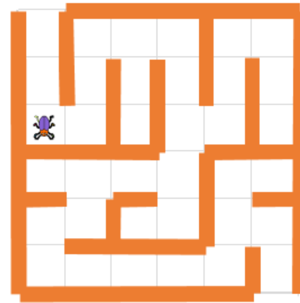
© ORB Education Visit http://www.orbeducation.com for the full, editable versions.

Robot Maze in Scratch



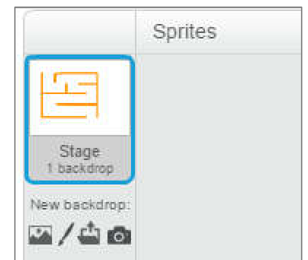
In our first set of Algorithms resources, we developed some standard instructions for escaping a maze. We'll now have a go at creating this in Scratch.

This project is difficult; there are lots of little things to get right. Use the tips below to help you through, but add your own personal flourishes wherever you like. Aim to make your finished project look different to ours so that everyone can see you have built it yourself.



Aim: *Using Scratch programming to create a maze escaping robot.*

Designing Your Maze

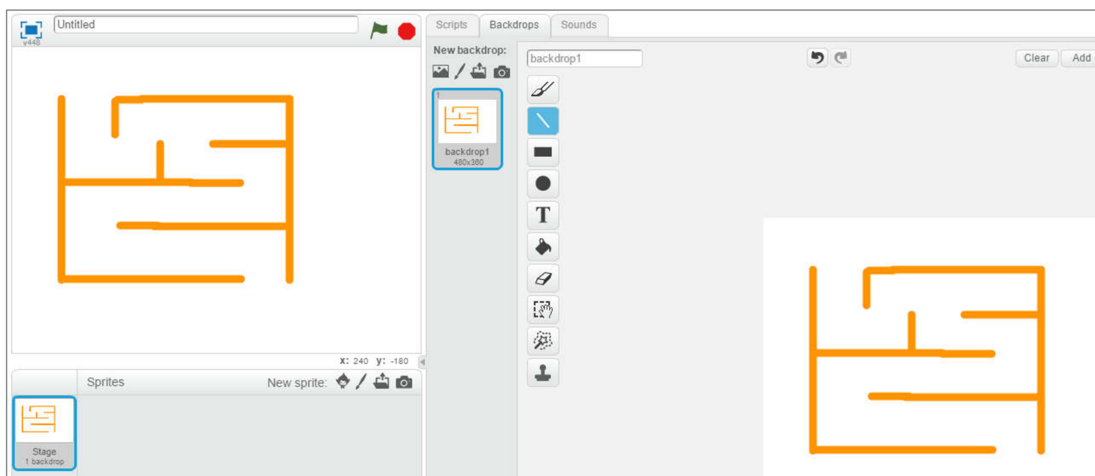


a. Like ours, yours should be a simple maze. This means that there is an entrance, an exit and all the dead ends you like, but no places where the robot can go around in a circle. There should be no loops.

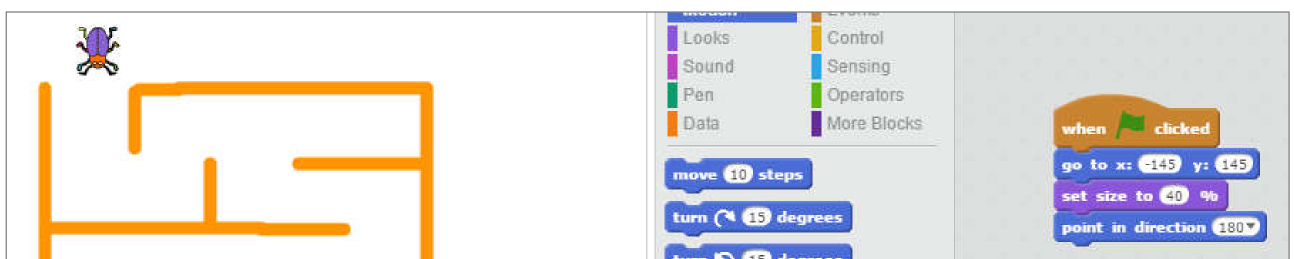
b. Your maze will be the backdrop to the stage. The backdrop is set using the controls to the left of the Sprites pane (as shown on the right).

c. Start by creating a simple maze using the painting tools in Scratch. Click the 'Paint new backdrop' icon and use the whitespace on the right as your canvas.

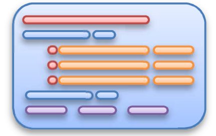
d. Choose the line tool, increase its thickness and start creating your maze. Hold down the shift key whilst drawing so that the lines are perfectly horizontal or vertical. Use the keyboard shortcut 'Ctrl + Z' to undo the last line. Make your first maze very basic.



e. Select a sprite to use as your robot. We have chosen a beetle but any will do. Use a script like the one shown below to position your robot in its starting position. You will need to set it to a suitable size (ours is 40%).



© ORB Education Visit http://www.orbeducation.com for the full, editable versions.



Getting Started with Python

As you have learned, there are many programming languages out there. We are going to have a look at some Python programming code. The reasons for choosing this language include:

Aim: *Try some Python programming.*

1. *It is relatively easy to understand – there are not too many fiddly rules to remember.*
2. *Python is a cross-platform language. It can be used on many devices and operating systems.*
3. *You can create and test some simple scripts using the website www.repl.it.*
4. *There are a large number of libraries available online. These are ready-written programs that carry out common tasks.*
5. *Python is used by many of the biggest tech companies.*

Having said this, programming languages have many similarities. Once you've got your head around the basics of programming it's much easier to learn new languages as and when you need to.

Task 1 – Getting Started in Python Online

You will be using a website called *repl.it*. This site allows you to play around with bits of Python code without the need to install applications. There are limitations to what you can do in *repl.it*, but it's a great way to get started.

a. Navigate to the website https://repl.it/languages/python3 (or visit www.repl.it and search for Python3). It is better to set up an account so that you can save your work.

b. In a new *session*, type the code **print ('Hello, world!')** in the coding window. This is the white window on the left of the screen.

```
1   print ('Hello, world!')
```

c. Click the *Run* button (or press 'Ctrl + Enter'). The Run button briefly turns to *Stop*. This can be used to halt programs.

d. Look at the output in the console. This is the black window on the right of the screen.

View the output from the program

```
Python 3.5.2 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
›
Hello, world!
›
```

e. Change the code in the coding window so that the output displays a different message when you run the program.

```
Python 3.5.2 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
›
This is a different message.
›
```

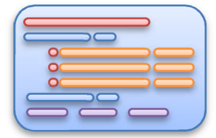
f. Create code so that you get the output shown on the right. You can use two lines of code with two print statements.

```
Python 3.5.2 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
›
Message 1
Message 2
›
```

Extension

Try achieving this last task with one print statement. Use **\n** to create a new line in your text.

© ORB Education Visit http://www.orbeducation.com for the full, editable versions.



A string is any sequence of characters such as letters, numerals, symbols and punctuation marks. A string has a certain length and with a little bit of coding, can be divided, reversed or changed to uppercase etc. We can operate on strings using functions and methods.

***Aim:** To investigate the manipulation of strings in Python.*

Task 1 – String Functions

A function is a section of programming code that performs a specific task. It is common to enter some data into the function (the input) and receive some different data back (the output). There are lots of built-in functions in Python that you can make use of. You can also write your own functions to carry out a particular job.

The syntax for using a function with a string is: *function(string)* e.g.: `len(my_string)`

***Note:** If you have worked through all tasks up to this point, you will have used the built-in functions **input**, **int**, **str**, **float** and **print**.*

len is a built-in function which tells you the length of a string.

len("rose") is 4, as there are 4 letters in the word "rose". Notice that the word *rose* has to be placed in quotes in the code.

If a variable *my_var* is assigned the word "horse", then *len(my_var)* is 5. We don't need to place quotes around a variable in the code.

Set up the program on the right in *repl.it* and see how it works. Save as '***11.1 len***'.

```
1  input_text = input("Enter some text.")
2
3  text_length = len(input_text)
4
5  print("Your text has " + str(text_length) + " characters.")
```

Task 2 – String Methods

A method is in many ways similar to a function. In technical terms, it is a section of code that acts on a certain object such as a string. Both functions and methods are examples of *procedures*.

The syntax for using a method with a string is: *string.method()* e.g.: `my_string.upper()`

upper is a method that converts a string to uppercase letters.

```
1  text = "Hello"
2
3  output = text.upper()
4
5  print(output)
```

a. Set up the program on the right in *repl.it* and see how it works. Save as '***11.2 upper***'.

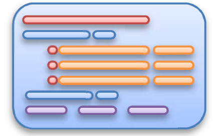
b. Write down the output you get from this program. ____________

c. What happens if you add some numbers and symbols to the original string? ____________

d. Create similar programs that demonstrate each of the methods in the table below. What does each method do?

Method	Save As	Example Use	Example Inputs	Outputs
lower	11.2 lower	output = text.**lower**()	"Hello", "HELLO"	
capitalize	11.2 capitalize	output = text.**capitalize**()	"hello", "HELLO"	
islower	11.2 islower	output = text.**islower**()	"hello", "HELLO", "Hello"	
isalpha	11.2 isalpha	output = text.**isalpha**()	"Hello", "Hello1", "111"	

© ORB Education Visit http://www.orbeducation.com for the full, editable versions.



Aim: *To set up and edit a list.*

A list in Python is an ordered set of data. The following line of code will create a list called *my_list*. In this case, the list will contain integers.

```
my_list = [12, 22, 351, 7812]
```

The list has four items (or elements). You can find what data is in any position in the list using the index. The indexes start at 0.

`my_list[0]` *This is 12, the first item in the list.*
`my_list[1]` *This is 22.*
`my_list[3]` *This is 7812, the last item in the list.*

Task 1 – Investigating Lists

The program below creates a list of numbers set as the strings "One", "Two", "Three" and "Four". When the program is run, the console displays the output shown on the right.

```
List created
Checking list in order: One
Checking list in order: Two
Checking list in order: Three
Checking list in order: Four
The 1st item in this list is: One
The 4th item in this list is: Four
One
Two
Three
```

a. Recreate the program in *repl.it*, saving as '**13.1 Basic Lists**'. Use the code and the output to work out exactly what each line is doing and add comments to the program to demonstrate your understanding.

Note: *This program uses 'for' loops, a type of count-controlled loop.*

```
1  numbers_list = ["One", "Two", "Three", "Four"]
2
3  print("List created")
4
5  for elem in numbers_list:
6    print("Checking list in order: " + elem )
7
8  print("The 1st item in this list is: " + numbers_list[0] )
9  print("The 4th item in this list is: " + numbers_list[3] )
10
11 for i in range(0, 3):
12   print(numbers_list[i])
```

Run through each element of the list and display it. (lines 5–6)

Find the data in any position using the indexes, which start at 0 for the first element. (lines 8–9)

Start the variable i at 0 and add 1 each loop until you reach 3. (lines 11–12)

b. There is a problem with the last section. It's only displaying "One", "Two" and "Three". This is because the range (0,3) only includes the numbers 0, 1 and 2; it stops short of the last item. Change the code so that it displays the whole list.

c. Edit the code so that the list includes numbers up to "Six". It should also display all six numbers when running the last section of code.

d. Add a line that writes "The 5th item in this list is…" to the console, as has been done with the 1st and 4th items.

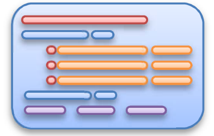
e. Add some blank lines to separate the different sections in the output. Use the code: `print("")`

f. Add a section which displays the number of items in the list. Use the code: `items_int = len(numbers_list)`

g. When you added items to the list, you would have had to change the last number on line 11 (in our original code above). It would be better for this to use a variable so that the change is automatic. Try the code on the right. Save your work.

```
for i in range(0, items_int):
```


© ORB Education Visit http://www.orbeducation.com for the full, editable versions.



Some procedures perform a set of tasks without returning a value. In other programming languages, these might be called subroutines. In Python, they are all **functions**.

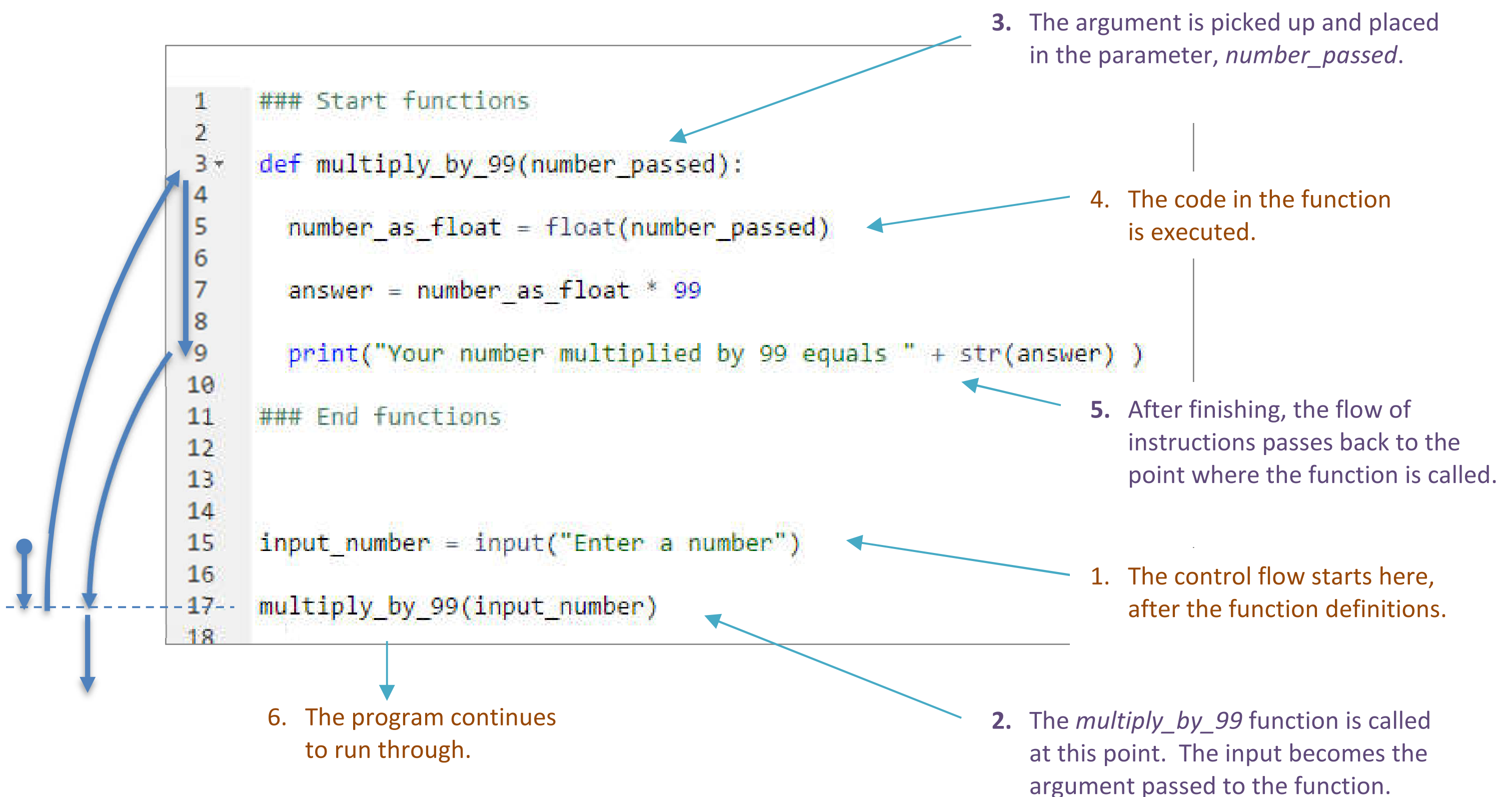
***Aim:** To learn how to create and call procedures.*

Task 1 – Calling Functions

The program below was created in *repl.it*. Because of the way Python works, any functions that are used have to be defined before they can be called. For this reason, we have placed ours at the top of the code.

The actual program starts on line 15 where the user is asked to input a number. A function named 'multiply_by_99' is then called. As you might guess, this multiplies the user's number by 99 (using the star sign for multiply).

***Note:** To keep things tidier, functions can be placed in separate files and imported at the start of the program. We used this idea when importing the math and random modules.*



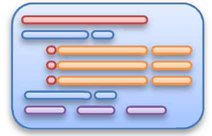
a. Create the program above in *repl.it* and add comments to the code explaining how each part works. Save the program as '**18.1 Functions A**'.

b. Copy the code to a new repl and adapt it so that it asks for two numbers from the user. It should then send the numbers to a function to be multiplied together (called *multiply*). Change anything that you think needs to be different and include comments to explain what you are doing. You will need to pass two arguments to the function's parameters as shown below. Save as '**18.1 Functions B**'.

```
def multiply(number1, number2):
```

```
multiply(input_number1, input_number2)
```


© ORB Education Visit http://www.orbeducation.com for the full, editable versions.



We're going to create a simple game that involves a player guessing the location of some hidden treasure in a grid. Once you have built the game, you can develop it in any way that you choose.

***Aim:** To use our ideas and programming skills to create a game.*

Task 1 – The Basic Game

As the programmer, it's your task to create the game for your friends to play. We will start by hiding treasure in one spot on a very small 3x3 grid. Once the program is ready, the player guesses the location and is told whether it is correct or not. If it isn't, then the player tries again until the treasure has been found.

As an example, the location of the treasure in the grid on the right is **B3**.

	1	2	3
A			
B			X
C			

List named *playing_grid*

	0	1	2
0	0	0	0
1	0	0	1
2	0	0	0

From a programming point of view, the grid will be stored in a two-dimensional list as shown on the left. All the elements of the list will contain a '0' except the one with the treasure, which will be a '1'.

```
playing_grid = [[0, 0, 0], [0,
```

Here are some of the challenges you will need to overcome:

- Creating the list. To start off with, you can manually set one element to '1' and the rest to '0'. This is partly shown above.
- Allowing the player to input a guessed location such as B2 or C1.
- Converting this location into a list element such as *playing_grid [1][1]* or *playing_grid [0][2].*
- Checking to see if this location contains a 0 or 1.
- Offering some output.
- Repeating until the 1 has been found. A *while* loop and a Boolean variable will allow repeated guesses.

a. Write some pseudocode and draw a flowchart for this set of instructions.

____

____

____

____

____

____

____

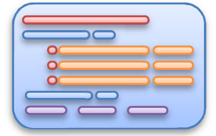
____

____

____

© ORB Education Visit http://www.orbeducation.com for the full, editable versions.

Creating a Game (page 2)



h. Have a go at creating the game in *repl.it*. Save this version as '**21.1 Basic Game**'.

- Declare and populate the two-dimensional list with dimensions 3x3 (as partly shown on the right). Remember to make one element a '1' and the rest '0's.

```
playing_grid = [[0, 0, 0], [0,
```

- Accept the user's guess and split this into two parts using slices like the one shown.

```
row = guess[0:1]
```

- Find the 1st index of your array by converting the letter from the player's guess. You could use an *if* statement something like the one on the right. Remember that *elif* is Python's way of saying *else if* and that == means '*is the same value as*'. A single equals sign changes a variable to the given value.

```
if row == "A":
    row_index = 0
elif row == "B":
    row_index = 1
```

- Find the 2nd index of your array by converting the number from the player's guess. A simple subtraction should work, but you'll need to change the text number to an integer number first (remember that the text "1" is not the same as the number 1).

- Create some output that tells the player whether they have found the treasure or not. If your loop is set up correctly with a Boolean variable condition, then the player should be able to keep guessing until the treasure has been found.

```
is_treasure_found = False

while is_treasure_found == False:
```

```
Enter a location from A1 to C3. A1
Try again.
Enter a location from A1 to C3. B2
Try again.
Enter a location from A1 to C3. B3
You found the treasure.
```

Task 2 – Improving the Game

Once your game is up and running in its most basic form, there are hundreds of ways you can improve it. Start with the suggestions below but incorporate your own ideas wherever you think they will help. Save this version as '**21.2 Improved Game**'.

The User Interface

- Make sure that the input is not case sensitive, so 'b3' is the considered the same as 'B3'.

- Add spacer lines so that the output is more readable.

- Add a counter that shows how many turns the player has had. Use *count +=1* as shorthand for *count = count + 1*.

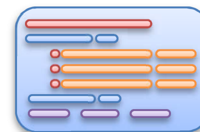
- Create a validation rule so that the player can only enter coordinates between A1 and C3. You could start off using an *if* statement with plenty of logical operators like the one below, but you'll need to improve on this method if the grid is enlarged at a later time.

```
Turn 1
Enter a location from A1 to C3. D3
Your location is invalid.

Turn 1
Enter a location from A1 to C3. A3
Try again.

Turn 2
Enter a location from A1 to C3. B3
You found the treasure.
```

```
if(row!="A" and row!="B" and row!="C") or (col!="1" and col!="2" an
    print("Your location is invalid.")
    print("")

else:
```

- Add any other bells and whistles that you think improve the interaction with the player.

Creating a Game (page 3)



The Game Board

- Move the creation of the 2D list into a separate procedure to keep things tidy. This also gives you the possibility of placing the function in a separate file at a later time. You will need to declare the list at a global level so that it can be accessed across all functions.

```
def create_grid():

  global playing_grid
  playing_grid = [[0, 0, 0], [0, 0, 1], [0,


create_grid()
```

- Try the code below-left for printing out the list line by line in the console. It is good for visualising the grid before playing. You don't want to give the location of the treasure away though, so set all the elements to 0 first then change a chosen location to 1 after printing the output.

```
for i in range(0,3):
  print(playing_grid[i])
```

```
playing_grid[1][2] = 1
```

- We've fixed the coordinates of the treasure whilst testing, but a great next step would be to allow you as the programmer to input these at the start of the game. You can begin working on a solution for this yourself, but you should soon find that you are repeating code that you have written before. For example, you will need to validate the programmer's input in the same way you validated the player's. It is time for some **Modular Programming**.

Task 3 – A Modular Approach

We're going to work on a modular solution (one that is split into sections using functions) so that firstly, the code remains neat and tidy, and secondly, we don't have to repeat sections of code. Make a copy of your program and save this version as '**21.3 Modular Game**'.

This is an overview of what we want to happen:

1. Create the grid and fill with 0s.
2. Let the programmer input the coordinates of the treasure and validate this input. Repeat until a valid location is entered.
3. Set the validated input as the location of the treasure.
4. Let the user guess the location. Validate the input and repeat until a valid location is entered.
5. Repeat the previous step until the treasure has been found.

a. Have a go at creating a flow chart for this process. Include as many details as you can.

b. We are going to use the following three functions:

create_grid()

validate_entry()

play_game()

The *create_grid* and the *play_game* functions will both use the *validate_entry* function to check that the input is acceptable. This is the case whether the programmer is entering the location of the treasure or the player is guessing.

© ORB Education Visit http://www.orbeducation.com for the full, editable versions.