**TEACHER INSTRUCTIONS / SCRIPT:**

**This document contains some optional information about the assignment and more barebones instructions for mentors, with a few notes about optional areas to expand with explanation.**

Prerequired knowledge:

* This is not intended to be delivered as a first intro to python, but not impossible to complete even for a first timer.
* Experience in scratch or other block-based coding language is useful.
* Lesson designed for students who have already been learning python for a few weeks and want to get some milage using the more difficult programming constructs and seeing how they fit together but can be delivered to other students too.
* Ideally learners have at minimum already written a hello world style program, so have had a brief description of the print function and variables. Also, helpful if they have come across arithmetic operators and functions.

Specifications:

* lesson should take about 1 ½ hours. (including a short break in the middle)
* For age range P6 – S3
* Requires a laptop or computer.
* Code should be run in the Raspberry Pi Foundation Code editor, they’ll need an account to save their work.
* Designed to be run in an afterschool coding club i.e. CoderDojo
* This tutorial uses the P5 graphics library, it can be a bit janky and the documentation is quite bad -> but all you need to know is that “run()” calls “setup()” then infinitely loops over “draw()”, you could try to write pseudocode out for this for particularly keen students.

Teaching notes:

* When introducing a coding concepts/construct may be helpful to write a toy example in the terminal before using in real code.
* Could be delivered by a mentor acting as a ‘teacher’ or through handing out printed instructions. (or both)
  + If it’s too many pages to print students can always pull up a Google Doc of the instructions although this means they can’t tick off tasks as they go.
* Mentors should let students follow sheet and encourage them to solve problems on their own.
* When a Mentor does intervene to provide help due to a hard problem, or difficult bug they should try to first provide more information to try to draw the attention of students to the problem.
  + Try not to give answers straight away 😊
  + When the student needs to write a new line of code or edit a line, have them do this themselves, don’t take control of their keyboard!
* Could be delivered in a pair programming style but this may be unsuitable depending on the environment (I’ve found that students are there to also learn how to type so having a driver and typist may not be a good idea)
* If presenting it may be useful to write comments in code, you could even encourage students to write comments as they go.
* If quizzing a student on a concept previously explained earlier in the session, try to wait some time before doing this rather than asking immediately after the explanation. (although you could optionally choose to do it then as well as later)
* This lesson could be delivered in a spaced repetition format if time allows it.
  + 20 learning + 10 unrelated physical task + 20 min learning + 10 min unrelated physical task + 20 min learning + 10 min showing off what you’ve done! (1 ½ hours)
  + Unrelated task could be….
* Where possible encourage students to use common programming shortcuts
  + copy/paste, tab, shift+tab, commenting out blocks of code with ctrl+/
* Analogies are super useful when it comes to explain abstract programming concepts but remember to also highlight the differences between the analogy and the thing you are explaining.
* Students tend to skip explanations written in the tutorial sheets and focus only on the tasks, so it is important for mentors to provide further explanations if they are especially important.
* Instructions include parts where students predict outcome of code, mentors should facilitate this as it will most likely be skipped otherwise.
* Assignment has some tasks without code snippets, use these to gauge how well students are absorbing lesson before the quiz at the end when there is still time to explain.

Python Concepts being used: (in bold are the main concepts being covered / focus for the tutorial)

* variables
* booleans
* **global variables**
* functions
* **if statements**
* nested if statements
* RGB values and some vector graphics
* taking user inputs
* integers
* arithmetic operators
* *Game design*

Setting up environment:

* Students need to access the Raspberry Pi Foundation Code Editor, ideally they make an account so they can save their work, but it is still possible to complete without an account.
* Once they have it open, instructors should copy and paste the finished code into their environments so they can see what the final product looks like
* After this, mentors should replace this with the skeleton code for the students to fill out themselves.

**Teacher / leader instructions:**

[Before beginning session, let students see finished game, so they know what they are creating, ideally, we let them run it themselves]

1. **Set screen size.**
   1. Add global variables width and height = 800 and 450.
      1. Variables refresher / introduction
      2. Describe global variables vs local variables.
   2. Filling out “setup()” (should be predefined in skeleton).
      1. What is a function?
      2. It can be helpful to explain that def stands for define.
      3. Why are functions useful?
         1. Stop you having to rewrite code.
         2. Could use a separate toy example in terminal to display this feature.
      4. Common issue when working with functions is putting a space in between words instead of an underscore, it would be useful to mention that you can’t use spaces when naming functions.
   3. In setup(), set “bgd\_colour to Color(255, 255, 255)” (white)
      1. (optional) Explain RBG values and let students see the effect of changing these values on the colour of their background. Note: they can’t have a black background, as it will make game unplayable…
   4. Below that write “size(width, height)”
   5. Add “run(frame\_rate=15) right at the end of the program…
      1. We will explain the frame\_rate parameter more later when it’s effects are more noticeable.
      2. For particularly keen students refer them to pseudocode / further explanation on handout for what run does in the context of the p5 package…
   6. Add comments describing what these functions just did.
      1. Ideally let students do this themselves as this helps with the learning process.
   7. Give students some time (5 min) to change the values here and observe effect.
      1. Mentors should know the default values, so they can revert changes if student forgets what original values were.
2. **Draw player!**
   1. In global variables (player related vars), set player\_x and player\_y to 10 and 200.
   2. Def draw\_player() below draw()
      1. Possible time to quiz students on why we use functions.
   3. Global player\_x, player\_y in function
      1. Explain why this is needed.
         1. Because we are going to edit this variables need the changes to happen on a global level!
   4. player\_colour = Color(255,0,0) (red)
      1. Let students choose their own colours!
         1. May be useful to have a reference sheet to hand with common colours, or link to a website with an RGB colour picker.
   5. fill(player\_colour)
   6. player\_height = 20
   7. player\_width = 10
   8. rect(player\_x, player\_y, player\_width, player\_height)
      1. students can play with changing width and height, but these are good defaults, so won’t cause headaches for mentors trying to debug student’s code.
      2. Could be useful to point out that rect stands for rectangle and give a brief description of parameters it takes.
   9. Add draw\_player() to draw() function, and run program! (a rectangle representing the player should appear)
3. **Make** **player** **fall**!
   1. In global player related variables, define falling = True
   2. (optional) Discuss in class how we could try to work out if there is a platform beneath our player.
      1. By getting the colour of the pixels directly below our player’s y position
   3. In setup(), write global bgd\_colour at beginning of function
   4. In draw\_player(), write “current\_colour = get(player\_x, player\_y+player\_height+1)”
      1. Explain what this function does, particularly why we have chosen these parameters. There is commented code in the skeleton which should help.
      2. Briefly (it’s not very important) explain that hex codes are just codes we can use to refer to RBG colours.
      3. (optional) to test understanding you could call this variable “temp” at first and then have students collectively or independently come up with a suitable name for it.
      4. Common mistake here is seeing the plus to the y value and thinking this means up, while in this context it means further down the screen
   5. Write “If contact\_colour.hex == bgd\_colour.hex” below this.
      1. Explain if statements using terminal
      2. Explain using == and !=
         1. Could possibly arrange an if using a != and negate to show that it does the same as ==
      3. Now return to our If statement and explain what it does, and why
   6. Inside if statement write 4 lines:
      1. # falling!
      2. player\_y = player\_y + 5
         1. (optional) show that the alternative way to write this is player\_y += 5, and that it does the same thing
      3. print(“falling!”)
      4. falling = True
   7. else:
      1. # no longer falling
      2. falling = False
   8. add “elif contact\_colour == Color(255,0,0).hex:”
      1. inside this write “# dead!”
      2. and “print(“you fell in lava, game over!”)
         1. students can write whatever game over messages they want.
   9. run and test.
      1. we should land on a platform and stay on top of it
4. **Game Over!**
   1. define a new global player related variable called “game\_over” and set to False
   2. add game\_over to global variables in draw\_player()
   3. in the elif of the draw player if statement, write “game\_over = True”
   4. In Draw(), create another if statement to only draw\_player() if gameover == False
   5. Test by setting player\_x in global vars to 60, and running
      1. should see player fall into lava and game over message printed
5. **JUMPING**!!! This is the longest / hardest part.
   1. We are gonna use the predefined jump function… You can walk students through it if you want or let them try to reason what it does collectively, but it may not be the clearest so you can just give a brief description of what it does if you don’t want to dwell too long.
      1. important to note that jump works in 10 steps, so that there is a nice animation rather than just teleporting the player to the next platform in 1 step.
   2. Define jumping global variables.
      1. jump\_distance = 0
      2. jumping = False
      3. jumps\_left = 0
      4. jump\_progress = 0
      5. jump distance is how far we want our player to jump and jumps left is how many jumps we want our player to make.
      6. jumping is a Boolean value to show
   3. still in global variables section set jump distance and jumps left using int(input(“how…. ?”))
      1. – 1 from jumps left as in python we start counting from 0
      2. you will also need to add jump\_distance, and jumps\_left to global vars
   4. add jumping, jump\_progress, jumps\_left to globals in draw\_player()
   5. if not falling and not jumping (in draw\_player()):
      1. jumping = True
   6. set falling if statement to “if contact\_colour.hex == bgd\_colour.hex and not jumping:” so we don’t fall while we are in the middle of a jump
   7. elif jumping == True:
      1. if jump\_progress < 10:
         1. it’s called a “nested if” when an if is placed inside another if like this.
         2. jump(jump\_progress)
         3. jump\_progress += 1
            1. explain that the implementation here, mainly jump\_progress is to make a jumping animation

could reference game design here, “while it would be easier to just update our position in one go this would look kind of ugly to a user of a game. Considerations like these are important in game design! “

or something along those lines

* + 1. else:
       1. if jumps\_left > 0:
          1. jumping = False
          2. jump\_progress = 0
          3. jumps\_left -= 1
          4. This code handles when we run out of jumps.

1. **Win** **Condition**
   1. turn students attention to the check\_if\_finished() function
   2. Explain how it works.
      1. get colour at ‘head’ height of player
      2. if the colour is green then we have reached the finish
      3. else return False (we have not finished)
         1. Could explain function returns here or leave for a later lesson. (students will probably not learn what it means to return a function here but this will give them a bit of exposure at least)
   3. in draw():
      1. add finished to global vars
      2. if not finished:
         1. draw\_level()
         2. finished = check\_if\_finished()
         3. draw\_player()
      3. else:
         1. text\_size(25)
         2. text\_align(CENTER, CENTER)
         3. text(“you WIN!”, width/2, height/2)
   4. run code – what are we missing??
   5. program man to walk when out of jumps:
      1. at the end of draw\_player()
      2. if jumps\_left <= 0 and not falling:
         1. (if not out of jumps and not falling)
         2. jumping = False
         3. player\_x += 2
   6. Should now have working game!
2. **Tuning** **and** **tweaking** **and** **playing**:
   1. have students complete the game.
   2. Have them investigate what happens if framerate is changed.
   3. If extra time or for extra fast workers, you can have them change the platforms of the level by editing the for loop in draw\_level, or changing some of the games global variables to see the effects!
3. **QUIZ**!
   1. B
      1. If statements
   2. HEY GUESS WHAT?, I LOVE FUNCTIONS, Oh yea I do….
      1. Calling functions
   3. 55
      1. Global variables + tracing algorithms
   4. hello!
      1. Global variables vs. local variables
      2. This one is the trickiest… the previous question should help answering though.

**Lesson 2 plan: TBC**

* make all player actions programmable.
* have students design their own harder levels to complete, or provide predesigned ones to find solutions for.
* if people finish these then have them extend the game by changing properties
  + could add new features such as a meteor shower to make game harder / different.