KS5-HE Transition Guide

Checkpoint Task

Instructions and answers for teachers

These instructions should accompany the OCR resource 'Types of programming language'
KS5-HE Transition guide *which supports OCR A Level Computer Science.*

Types of programming language

Activity 1a: Average of an array

Create a program that can be described in structured English as follows:

The 'Average Height' program:

Step 1: User enters the height in cm of their class into an array (list in Python)
Step 2: Array average is calculated by finding the sum of the array and the number of elements in it.
Step 3: The result of the calculation is output
Step 4: User is asked if they want to repeat the program

Each step is a procedure. There should also be a main procedure that calls the other procedures. The procedures need to pass the data along, so learners might want to have global variables declared either outside any procedure or on a more advanced level; they might want to use parameter passing. In the latter case, Steps 1 and 2 are functions, while Steps 3 and 4 are subroutines (don't return values).

Learners will need to initialise an array (or a list in Python) and append values to it using a *while* loop in Step 1.

Then, in Step 2, they should have a *for* loop that iterates through all array values and adds them up, while also counting the number of elements. Some languages might have builtin functions for sum of array (Python) but learners might want to do it the long way to demonstrate their mastery of iteration.

Step 3 is straight forward. It should use a value returned from Step 2.

ABC – This activity offers an opportunity for English skills development.

123 – This activity offers an opportunity for maths skills development.

In Step 4, learners might want to have a *while* loop for the user interface purposes, perhaps, presenting a user with a question if they want to calculate another average.

Activity 1b: Redo your program from 1a using functions (subs that return values) for all procedures except main ()

Activity 1c: Add a feature where users can add-in extra heights which are appended to the array/list.

Major points to watch out for:

- A colon at the end of a Python line indicates a four-space indent on the next line.
- Variables don't need to be specifically declared but it's a good practice to do so. Just initialise them (set the default value) before you actually use them and this will be as good as declaring them.
- Lists are arrays with easy functions like append, remove, len, index, etc.
- Lists use square brackets, functions use round brackets.
- Variables input by the user or read from files are strings and need to be 'cast' (converted) to numbers (int for integers, float for fractional, etc.) before they can be used in calculations.
- Try/except/else is used when casting and validating to prevent ugly crashes and steer the program around the possible crash, this is called error handling.
- Def <name> () : is used to create subs or functions.
- Functions must have return at the end which terminates the function.
- Levels of indentations can help you tell at a glance if a line belongs to a function, loop or both. For both it will be indented twice, once for being inside a function, and the second time for being in a loop.
- Procedural programming traditionally has a 'main ()' sub.
- Any sub can be turned into a function if you simply return an optional value (e.g. this could be a string 'sub xx finished successfully'). The rest of the program might not even use it. Higher-ability candidates can be challenged with creating a log of the program as it runs. It will write the return value of all functions to a text serial file to simplify troubleshooting, which is a very sensible idea.
- Global variables can't be modified from inside subs, unless they are declared inside subs with the word 'global' in front of them.
- It is possible to have a global and a local variable with same name, however, for Python they are different. By default, Python will think you are using the local copy, not the global if you have two such variables.
- Python uses identifiers of the format: word1_word2 for variables, functions, subs, WORD1_WORD2 for constants.
- Functions and subs can take parameters as a comma-separated list and functions can return a comma-separated list of parameters.

- Errors shown by Python often point to the mistake in the PREVIOUS LINE, e.g. a missing bracket will crash out on the next line.

String concatenation (joining) can be done both with plus and comma. Comma supplies a space (nice) and allows multiple data types to be combined in the same line of output; however, since tuples (comma-separated lists) are popular in Python, under certain conditions it will misinterpret your print statement to output a list, with unpredictable results.

Answers

Activity 1a: Using subs that don't return values

The following Python code:

```
global_my_array= [ ]     #global variables declared: an array and a number
global_average=0

def get_input () :      #a sub that appends the global array
    loc_user_num=0    #initialise a local variable
    while loc_user_num!=-999:    #rogue value will terminate input
          loc_user_num=float(input("Enter a number >> "))    #cast to float
          if loc_user_num!=-999:     #selection statement
              global_my_array.append(loc_user_num)    #add to the end of
array

def find_average () :
    global global_average     #global in front of variable allows to change
it
    #from inside the sub
    loc_my_sum=0     #yes, there is a built-in sum(list)....
    loc_counter=0
    for each in global_my_array:     #iterate through all array elements
        loc_my_sum=loc_my_sum+each
        loc_counter=loc_counter+1
    global_average=loc_my_sum/loc_counter     #famous average formula
sum/count

def show_result () :      #output
    print("The average is",global_average)    #concatenate with comma

def main () :
    loc_user_continue="y"     #default user intent
    while loc_user_continue=="y": #user interface loop
          loc_user_continue=input("Calculate another average? y/n >> ")

          if loc_user_continue=="n":    #exit on entry of "n"
            print("Bye...")
        else:
            get_input ()    #3 subs are called one after another
```

```
            find_average ()
```

show_result ()
main () #this triggers all of the code from above

Will produce this result:

```
Enter a number >> 12
Enter a number >> 13
Enter a number >> 14
Enter a number >> -999
The average is 13.0
Calculate another average? y/n >> y
Enter a number >> 34
Enter a number >> 35
Enter a number >> 36
Enter a number >> -999
The average is 35.0
Calculate another average? y/n >> n
Bye...
>>>
```

Activity 1b: Functions instead of subs

```
def get_input () :
    my_array= [ ]     #notice array is now local
    user_input=0
    while user_input!=-999:
        user_input=float(input("Enter a number >> "))
        if user_input!=-999:
            my_array.append(user_input)
    return my_array     #array is return and available to other parts of
prog

#print(get_input ())     #possible unit test of the get_input ()

def find_average(para_my_array) :
    my_sum=0     #yes, there is a built-in sum(list)....
    counter=0
    for each in para_my_array:     #iterate through the array
        my_sum=my_sum+each      #recursive addition
```

```
        counter=counter+1       #increment counter
  return my_sum/counter        #return average
def show_result () :
numbers=get_input ()      #numbers will become whatever get_input () function
#returns
    average=find_average(numbers)
    return "The average is "+ str(average)      #concatenate with a plus

def main () :       #bringing all code together, conditional loop for the
interface
    user_continue="y"
    while user_continue=="y":
        print(show_result ())
        user_continue=input("Calculate another average? y/n >> ")
        if user_continue!="y":
            print("Bye...")
main ()     #still need the main sub
```

Activity 1c: How would you carry out unit tests on your procedures?

Learners will attempt to run subroutines independently of main (), using print statements to see if correct values are returned.

```
#unit test of the find_average ()
print(find_average(get_input ()))
```

Activity 2a: Write a program that matches the following structure with comments that explain your code

Sub1
Sub2
Function1
Function2
Main

Answer: Most programs will have these procedures:
Sub1=read data, Sub2=show data, Function1=validate, Function2=compute/process

Activity 2b: If you were not constrained by this structure, how would you implement the same program? Can you think of a better structure?

Answer: Learners might talk about having more/fewer subs, not using functions at all, etc.

Activity 2c: A learner wrote a program with this very clever line in the procedure 'main':

"show_km(convert(validate(read_miles ())))"

Explain how the learner was able to do that and complete this program, with comments, by creating the procedures necessary to make this line work. Include a data flow diagram to illustrate how this procedure works.

Answer: Learners should be able to create something like this (Python3 used):

```
def read_miles () :      #here we don't set a variable to input, we return
it
    return input ("Enter distance in miles")

def validate(para_input) :
    try:     #preventing crashes if input is not a number
        ok_input = float(para_input)
    except:
        return 0
    else:
        if ok_input > 0:
            return ok_input
        else:
            return 0

def convert(para_dist) :
    KM_IN_MILES = 1.6     #constant identifiers are in CAPITALS
    return str(para_dist * KM_IN_MILES) + " km"

def show_km(para_data) :     #output
    print(para_data)

def main () :      #notice how functions lend themselves to onion layers
    show_km(convert(validate(read_miles () )))

main ()
```

Activity 2d: Rewrite the program in the imperative procedural paradigm with comments.

```
global_input=0     #initialise globals, you see the pattern - subs need
globals
global_result=0
```

```
def read_dist () :
    return input("Enter distance in miles")

def validate(para_dist) :
    #in Python subs need to be authorised to modify global variables
    global global_input     #authorise sub to modify global variable
    try:     #cautiously cast string input to float
        ok_dist=float(para_dist)
    except:
        global_input= 0
    else:
        if ok_dist>0:
            global_input= ok_dist
        else:
            global_input= 0     #invalid values are replaced with a zero

def convert () :
    global global_result
    KM_IN_MILES=1.6
    global_result=global_input*KM_IN_MILES

def show_result () :
    print(global_result,"km")

def main () :
    validate(read_dist ())
    convert ()
    show_result ()
    print("Bye...")
main ()
```

Activity 2e: Name the state variables used in this question. What is their usefulness? How could using state variables create problems? Provide an example of this situation.

Answer: The two global variables: global_input and global_result.

Their usefulness is in allowing different parts of the program (subprocedures) to talk to each other and share the information.

The problems that could arise are (a) difficulty in reusing this code in another program, which is a common practice among programmers, and if it is placed into a different program, the state variables are accessible to that program's other subs which could modify their values unpredictably.

Example: Program 1 converts litres to ounces and is already written, tested and ready. A programmer is starting on Program 2 which converts miles to km. Since both of them share similar user input and validation routines, the programmer decides to recycle the user input and validation parts of the Program 1 in Program 2, so both will use global_input and global_result as state variables. Additionally, since there are various types of ounces, in the Program 1 the programmer had to implement a state variable type_of_input which will accept a user's choice of a troy or metric ounce. If he/she recycles their input code, the state variable of type_of_input might not be set in the Program 2, and the calculations might produce unpredictable results. Another situation could be that more than one part of the program will modify global_result and if one part of the program expected it to be zero at one point and it wasn't, you could get random results. It can be said, as a general rule, it's best not to assume anything in programming.

Activity 2f: List the differences between your two programs.

Answer: Appearance of global variables, less parameter passing, easier to read.

Stretch exercises:

Activity 2g: Modify both versions of your program to ask the user the direction of the unit conversion, e.g. 'miles to km' or 'km to miles'.

Answer:

Functional –

```
def read_option () :      #create our first function to get user's choice of
units
    return input("Type 1 for Miles to KM; 2 for KM to Miles ")
#functions return values

def read_dist () :      #get the actual units, once we know they km or miles
    return input("Enter distance")    #all functions return values

def validate(para_dist) :    # cast string->float, if crashes return zero
    try:
        ok_dist = float(para_dist)    #local var to hold verified value
    except:
        return 0
    else:
```

```
if ok_dist > 0:     #validate for positive distances for input
            return ok_dist
        else:
            return 0     #return zero if a negative number was put in

def convert(para_dist, para_option) :
    KM_IN_MILES = 1.6     #constants are in capitals, they don't vary
    if para_option == '1':     #selection statement
        return str(para_dist * KM_IN_MILES) + " km"
#concatenation=joining
    elif para_option == '2':
        return str(para_dist / KM_IN_MILES) + " miles"
    else:
        return "Invalid data or conversion type"

def show_result(para_result) :
    print(para_result)

def main () :
show_result(convert(validate(read_dist () ),read_option () ))    #pass args
to     #function
    print("Bye...")

main ()
```

Imperative –

```
global_input = 0     #init globals
global_result = 0
global_option ='0'     #user input is a string

def read_option () :
    #in Python subs need to be authorised to modify global variables
    global global_option
    global_option = input("Type 1 for Miles to KM; 2 for KM to Miles ")

def read_dist () :
   return input("Enter distance")
```


```
def validate(para_dist) :
   global global_input
   try:
       ok_dist = float(para_dist)
   except:
       global_input = 0
   else:
       if ok_dist > 0:
           global_input = ok_dist
       else:
           global_input = 0

def convert () :
   global global_result
   KM_IN_MILES = 1.6
   if global_option == '1':
       global_result= str(global_input*KM_IN_MILES) + " km"
   elif global_option == '2':
      global_result = str(global_input/KM_IN_MILES) + " miles"
   else:
       global_result="Conversion type not specified"     #catch no choice
made

def show_result () :
   print(global_result)

def main () :

   validate(read_dist ())
   read_option ()
   convert ()
   show_result ()
   print("Bye...")

main ()
```

Learners' answers might vary depending on their preferences, but generally, the imperative approach should be easier to modify as the data flows are shared through state variables, while the functional approach requires more careful 'routing' of data between the procedures.

We'd like to know your view on the resources we produce. By clicking on the 'Like' or 'Dislike' button you can help us to ensure that our resources work for you. When the email template pops up please add additional comments if you wish and then just click 'Send'. Thank you.

If you do not currently offer this OCR qualification but would like to do so, please complete the Expression of Interest Form which can be found here: www.ocr.org.uk/expression-of-interest