![Glasgow Caledonian University logo]

<u>Module Title: Programming 0</u>

Module Codes: M1I326719, M1I326726

Level: 1

Module Leader: Richard Holden

Lab / Practical Workbook

Table of Contents

Overview

This workbook is designed to help you understand and practice using Python code, including functions, and the 'main block' for testing code.

This workbook is intended to encourage beginners to start thinking about simple problems and, using Python code, represent solutions to those problems. You will also use a main block to test your code. The workbook mirrors the level of complexity students will encounter in the coursework and will therefore be good practice towards that.

Note that it is very common for two different people to write *different* code to solve the *same* problem, but with help doing so throughout the course, you should try and write code that is generally:

- clean
- organized and;
- understandable

> *"programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil."*
>
> Donald Knuth

> *"Programming is the art of telling another human being what one wants the computer to do."*
>
> Donald Knuth

Write your own code. Check your solutions with those provided later in the course. Check your own solutions with those of other students, too, after solving a problem independently. You might find it interesting to see the variety of approaches that other people take. This will help you learn and memorise.

The functions provided range from simple mathematical operations to more complex tasks. Generally, this also reflects the coursework where some exercises will be more complicated than others. If you cannot answer the more difficult ones, move on to ones you can start on. This will help you improve gradually and build confidence before tackling more difficult problems. Furthermore, that should be *'more difficult'* problems; different students will find different problems more or less difficult. By completing the workbook, you will have a strong understanding of how to write Python scripts, which is the groundwork you need, as a beginner, for future success.

Separate to this beginner-level baseline, if you are progressing (maybe you are even finding all the problems too easy?) then try to solve some of the other practical *challenges* set on this course. Hopefully you'll find the challenges interesting.

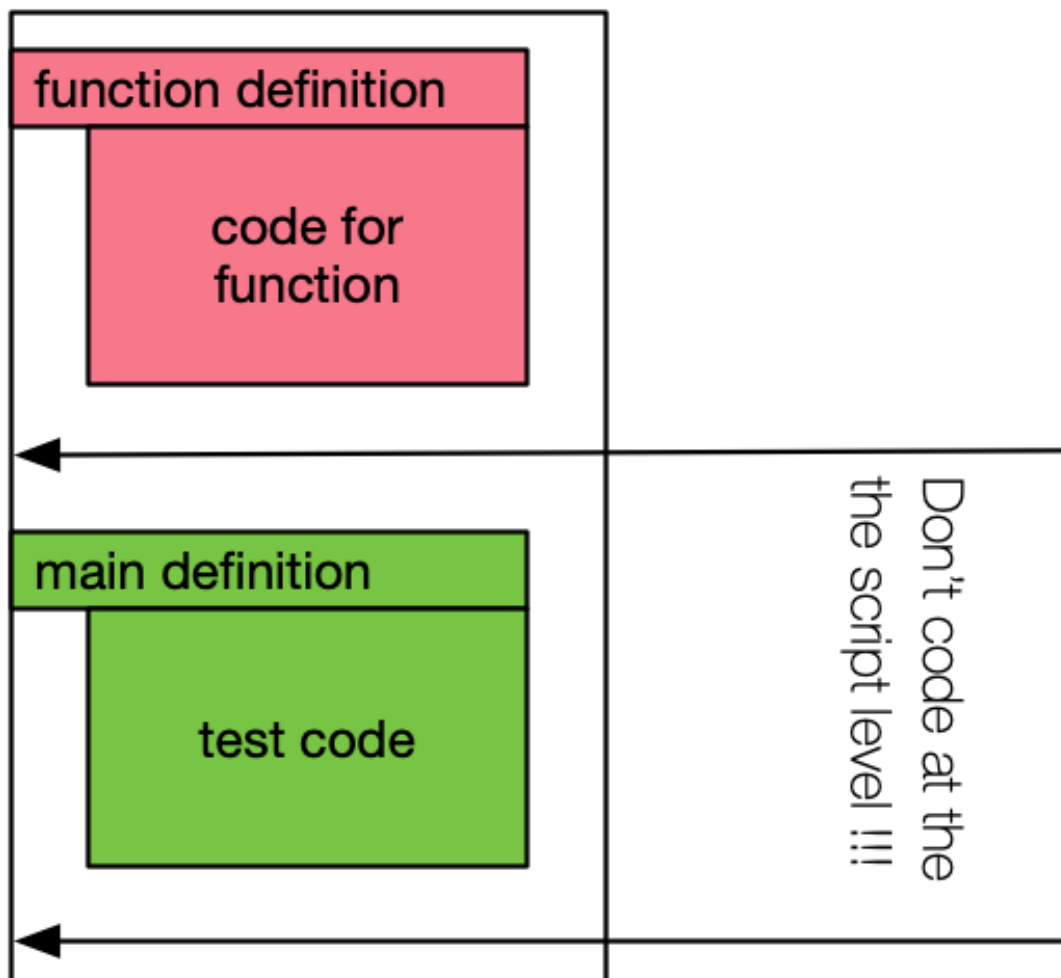# Script names and the 'main block'

Figure 1: General script structure schematic

```python
def function_name():
    # Example code that prints "Hello World" to the console
    print("Hello World")



# This runs when the script is executed directly
if __name__ == "__main__":
    function_name()  # Invoke function_name
```

Figure 2: General script structure code

Before we begin
What does the Python3 interpreter do with the script structure?

Briefly, the Python interpreter (the program that reads the code):
- **Defines the function**
- **Checks the script is being run directly**;
  - if so it **Calls the function** that prints `"Hello World"`.

In detail, the interpreter:

- **Defines the function. (`def function_name():`):**
  - When the interpreter encounters the `def function_name():` statement, it doesn't execute the function right away.It simply stores the function's definition in memory under the name `function_name` so it can be called later.

- **Checks the script is being run directly (`if __name__ == "__main__":`)**
  - The interpreter checks whether the special variable `__name__` is set to `"__main__"`. This variable is automatically assigned based on how the script is run: If the script is run directly (e.g., from PowerShell or Terminal), `__name__` is set to `"__main__"`.
  - If so **Calls the Function (`function_name()`):** If the script is being run directly (i.e., `__name__ == "__main__"` is `True`), the interpreter executes the code within the `if` block. This leads to calling `function_name()`, which triggers the interpreter to execute the code `print("Hello World")`.

Schema

In the provided workbook, students are expected to implement functions inside scripts. These functions will be explained in tables, below, based on a given schema. The schema contains two key pieces of information for each function:

- **Function name**: This specifies both the name of the script and the name of the function that the students need to implement. For example, if the function name is "adds(a, b)", it indicates that the function should be implemented to perform an addition operation between two parameters, a and b.

- **Description**: This is a natural language description that explains what the function should do. Based on this description, students are required to define the function signature and implement its logic accordingly.

The structure of the scripts is expected to follow standard practices where each script includes a main block. You will 'test' code from inside the main block.

Function Name	Description
This the name of the script and the name of the function in the script.	This is a natural language description and from it you must determine the definition and implementation of the function.

Example: adds function

Notice in the descriptions that it should **clear when function arguments are required** (phrases like receives/supplied/arguments etc. are good clues). For your coursework, I will be explicit about the function definitions. However, where there are none suggested in this workbook, try and work out yourself what these should be.

Function description

Function Name	Description
adds	Receives two inputs and returns the sum of those two inputs.

code solution

Note in the following 'solution', the function isn't very well tested. You will learn about how to test your functions more fully during the course. To begin with, however, printing the result to the console, in this way, will suffice.

```python
# Define the adds function
def adds(a, b):
    """
    This function takes two arguments, a and b, and returns their sum.

    Parameters:
    a (int or float): The first number to add.
    b (int or float): The second number to add.

    Returns:
    int or float: The sum of a and b.
    """
    return a + b

# Main block to test the function
if __name__ == "__main__":
    # Example usage of the adds function
    result = adds(10, 5)
    print(f"The sum of 10 and 5 is: {result}")
```
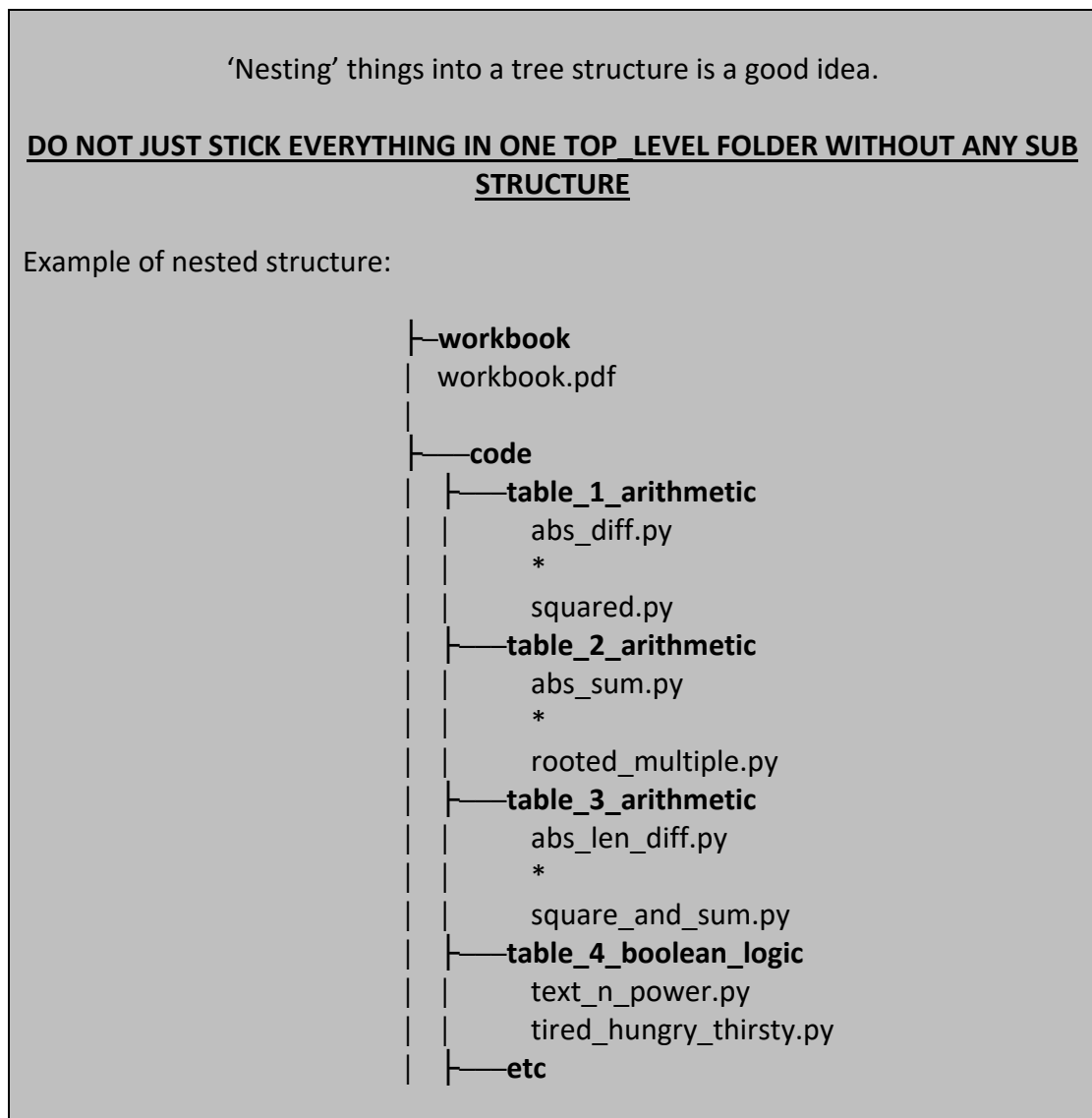
Figure 3: The adds function code example

Running the solution in the suggested environment:

Suggested workflow:

1. Have a specific **folder** to keep your scripts in. For example, say you have a \<workboo>k folder, with your workbook file \<workbook.pdf> in there, then another folder called \<code>, then a folder for each table listed in the workbook, that would be a bad start, then you can put the *scripts* in the appropriately-named folder:

'Nesting' things into a tree structure is a good idea.

DO NOT JUST STICK EVERYTHING IN ONE TOP_LEVEL FOLDER WITHOUT ANY SUB STRUCTURE

Example of nested structure:

```
├─workbook
│  workbook.pdf
│
├───code
│   ├───table_1_arithmetic
│   │       abs_diff.py
│   │       *
│   │       squared.py
│   ├───table_2_arithmetic
│   │       abs_sum.py
│   │       *
│   │       rooted_multiple.py
│   ├───table_3_arithmetic
│   │       abs_len_diff.py
│   │       *
│   │       square_and_sum.py
│   ├───table_4_boolean_logic
│   │       text_n_power.py
│   │       tired_hungry_thirsty.py
│   ├───etc
```

---

2. Open a PowerShell into that folder.
3. Implement (or reimplement) the solution.
4. Run the solution.
   a. If: code fails, go to step 3.
   b. Otherwise, go to step 5.
5. Admire your output.
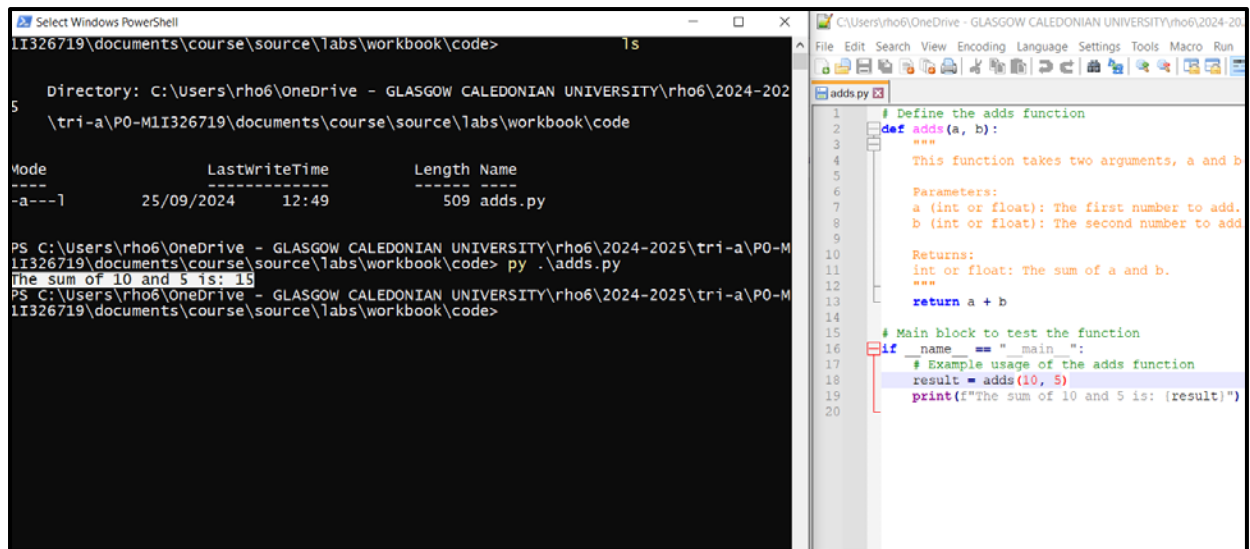
Figure 4: Simple coding environment

Function tables: Arithmetic

Function Name	Description
adds	Receives three numbers, adds them together, and returns the result.
multiplies	Receives list1 and list2, multiplies each element of list1, multiplies each element of list2. This results in two numbers. The function returns the sum of these numbers.
abs_diff	Receives two numbers and returns their difference as an absolute value.
squared	Return the square of a supplied number.
powered_root	Raise a supplied number arg1 to the power of arg2, then returns the square root of the result.
precedence	Receives 7 parameters a to g, which are arranged according to the following equation: $$\frac{(a + b \times c) \times d}{e - f} + g$$ …the resulting value is returned
arb_sum_powered	Receives a list. The sum of the first n - 1 elements is raised to the power of the nth element. The resulting value is returned.
rooted_sum	Receives a list of args. Returns as an integer the square root of the sum of a list of numbers.
multiply	Receives an arbitrary list of numbers (you can use *args, for example) and returns the result of multiplying those numbers.

Function Name	Description
adds	Receives a list of numbers, and returns the sum.
multiply	Receives five arguments and returns the result of multiplying those numbers.
multiplies	Receives list1 and list2. Each element of list1 is multiplied by the corresponding element in list 2 to produce a third list, the result of those element-wise multiples. Ensure that list1 and list2 are the same size, otherwise return an empty list.
abs_sum	Receives two numbers and returns their sum as an *absolute* value.
factorial	Receives a positive integer *n* and returns it's factorial. A factorial is the product (multiplication) of each positive integer less than or equal to *n*.
powered_multiple	Raise a supplied number arg1 to the power of arg2, then returns the result, multiplied by arg3.
modulo	The function takes two numbers as inputs and returns the remainder when the first number is divided by the second.
arb_sum_powered	Receives arbitrary arguments using *args. The sum of the first n - 1 elements is raised to the power of the nth element. The resulting value is returned.
rooted_multiple	Receives a list of args. Returns (as an integer) the square root of the result of multiplying each element in the list.

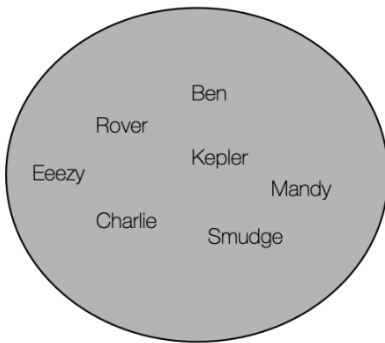| Function Name	Description
abs_len_diff	Receives two lists and returns the difference between their length as an absolute value, raised to the power of a number, which is the third argument to the function.
adds	Receives a list and returns the sum of the values stored at odd indices.
binary_multiply	Receives two binary [0, 1] lists and returns, as a decimal, the result of multiplying those two numbers. Use a for loop to decode the binary strings, which must be the same length.
elements_multiplied	Receives two lists of numbers of length m, and m * n, respectively. Values at index *i* of the first list are multiplied by values at index *j* in the second list, where *j = i * n.* Two examples are illustrated, here: You must implement the solution for <u>arbitrary values of n</u> and handle incompatible list sizes by returning an empty list.
multiplies	Receives list1 and list2. Each element of list1 is multiplied to obtain the product of the numbers. Each value of list2 is multiplied by the aforementioned product. Then, the sum of the elements in list2 is multiplied by the approximate value π (PI) = 3.14159 contained in the function.
multiply	Receives an arbitrary number of arguments (e.g. use *args, for example, as a function argument). The function should multiply the arguments together and return the string: "is_signed" …if the value is either negative or positive or: otherwise return the string: "unsigned"
precedence	Receives 7 parameters a to g, which are arranged according to the following equation: $$\frac{((a+b)*c*d/e - f + g)}{f} - g$$ Return the result of the calculation.
root	Return the square root of a supplied number.
smooth	Receives a list and returns a 'smoothed' version – i.e. a list where the value of each element is the average of itself and the left and right adjacent elements. 'Edge' elements at index 0 and index length – 1 should be averaged with their right and left adjacent neighbour, respectively.
square_and_sum	Receives a single list of numbers, takes the square of the values and stores those to a local list, then returns that list, but before doing so multiplies each element by the approximate value of the golden ratio ø (phi) contained in the associated script where ø = 1.618.

Function tables: Boolean Logic

Table 4: Boolean Logic

Function Name	Description
text_n_power	Consider a call to a script: text_n_power("three", 2) With if else statements, and covering command line input cases for stings {"one," "two", "three"} and integers {1, 2, 3, 4, 5} return the numerical value.
tired_hungry_thirsty	Consider three separate states – tired, hungry, thirsty. A table is provided, below. It is a mapping between states to an associated action. Create a function that takes three arguments, which are the states, and returns the corresponding action.

tired	hungry	thirsty	action
False	False	False	"do nothing"
False	False	True	"drink"
False	True	False	"eat"
False	True	True	"drink and eat"
True	False	False	"sleep"
True	False	True	"drink in bed"
True	True	False	"eat in bed"
True	True	True	"eat and drink in bed"

Function Name	Description
decimal_mapping	The numeric values on a Mac keyboard look like this: If the user holds the shift key while pressing key '6' the symbol '^' is produced. Assuming that the user either holds the shift key or doesn't, while pressing an arbitrary number of keys, create a function that receives the resulting list of key presses, as numeric values, and the boolean state of the shift key (is_pressed), then returns from the function the appropriate mapping from the list of numbers.
is_rule_30	This function accepts a list of lists. Each element list is a list of Boolean states of length = 4 and represent a single row in a boolean truth table. The function checks if this list of lists matches Wolframs 'famous' rule 30, and returns a True or a False Boolean value if it does or doesn't match. Rule 30 is represented in the table below: | Left (L)	Center (C)	Right (R)	Output
---	---	---	---
1	1	1	0
1	1	0	0
1	0	1	0
1	0	0	1
0	1	1	1
0	1	0	1
0	0	1	1
0	0	0	0	
do_the_dog	The script contains some lyrics from a song, *Do the dog*, by the band from Coventry, *The Specials*. The function receives a list of dog names. If the dog list contains some or all of the set of dog names specified here: Then the lyrics are returned. Otherwise the string "dinnae do the dog" is returned.

Function tables: Space

Table 6: Space

| Script Name	Description

black_hole	This function receives three arguments p1, p2 and r. Assuming that p1 is the center of a circle with radius r, then:
	
	if p2 is inside the circle, then return the string:
	
	"Beyond the event horizon!"
	
	if p2 is outside the circle, then return the string:
	
	"So far so good"
	
	If p2 is equal to the radius, then return the string:
	
	"Event horizon!"
	
	In this function you should assume that distance is measured as the Euclidean distance.

e_m_dist	Return either the Euclidean or Manhattan 2D distance between two points p and q, depending on if a third arg is "euclid" or "manhattan", otherwise return None.
manhattan_distance	Return the Manhattan distance given two lists of arbitrary dimensions, which the function receives .If the lists are different lengths return Pythons None value.
eucliden_distance	Return the Euclidean distance given two lists of arbitrary dimensions, which the function receives .If the lists are different lengths return Pythons None value.

Table 7: Space

Function Name	Description
is_overlap	Two 'circles' are received in respective lists containing coordinates x, y and radius r. Return True or False, depending on whether or not the circles touch or overlap.
neighbourhood_1	This function returns 1D Moore neighbourhood offsets as a list of lists.
neighbourhood_2	This function returns 2D Moore neighbourhood offsets as a list of lists.
neighbourhood_3	This function returns 3D Moore neighbourhood offsets as a list of lists.
neighbourhood	From the function return a string literal, which explains why general functions might be more preferable than specific functions when it comes to coding solutions to problems. 100 words.

Moore neighbourhoods in 1-dimensional and 2-dimensioanl space:

1-Dimensional

2-Dimensional
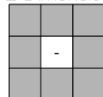
Assume that the 'Moore' neighbourhood offset does not contain the origin cell:
1-D this means the cell at index 1
2-D this means the cell at indices 1,1

What is an offset?

Offset means the *difference* 'offset' from the origin coordinate. For example, in a 1D Moore Neighbourhood, the required offsets are [[-1], [+1]], if the origin offset, 0, is not included in the list of lists.

Function tables: Strings

Function Name	Description
length_as_text	Takes a string of length in the inclusive range between 1 and 4, and returns a string (either "one", "two", "three", "four" or "error: string out of range".
concatenate	takes two or more strings as input and returns a new string that is the result of joining them together in the order they were passed.
upper	takes a string as input and returns a new string where all the characters are converted to uppercase.
lower	takes a string as input and returns a new string where all the characters are converted to lowercase.
quote	Receives a string, which you can assume is a quotation by someone. That someone is also received as a string. Return a single string with the quote on one line and the name of the 'someone' on a new line and three tabs to the right.

Function Name	Description
quote	third line under the quote and two tabs to the right. Use escape sequences to achieve this.
descriptions	Assume a function receives two strings. Each string contains some description, which has more than 5 words (this should be checked inside the function). If either one of the strings do not contain more than 5 words return a float of 0.0. Otherwise, return the number of words in the first argument, divided by the number of words in the second argument to the function.
is_palindrome	A palindrome is a word that reads the same backwards as forwards. Implement the function, which returns a Boolean answer (True / False).
lyric_count	Receives a string and returns the number of occurrences of a given word that matches the second string received as a parameter.
move	a function receives a discrete speed and acceleration (assume direction pointing right), then returns an update to some graphics that are also sent into the function. Example: speed = 2 acceleration = 3 graphics: ##P##################### Result: #######P################### i.e. the position P moved 5 steps forward. If the position P moves off the edge of the world, set P to position zero and have the graphics reflect that case.
the_types	Receives a list and returns a corresponding list containing the types of the variables held in the received list.