**Tivadar Debnar**

**ICON College Computing HND**

**Programming Assignment**

**Introduction**

**What are algorithms and their process in building applications?**

“*An algorithm is an explicit, precise, unambiguous, mechanically-executable sequence of elementary instructions, usually intended to accomplish a specific purpose*”. (Erickson, 1999)

We can design algorithms to solve a wide variety of problems, let it be as trivial as summing an array of numbers, or more sophisticated functionalities, like encryption, and graphic rendering.

A standard method for writing an algorithm for an advanced computing problem is ***decomposition***, where we break down the given task to several minor sub-routines to the point, we end up with a bunch of specific sub-functions that are easily solvable with clear, straightforward instructions.

Another common tactic is called ***abstraction***, where the programmer reduces all unnecessary details that pollute the function, so the cardinal steps to the solution will become more articulate.

Suppose, we describe the algorithm as a ***language-neutral*** representation of the implementation of a particular problem. Instantly becomes apparent that a significant advantage of algorithms is that the author is ***not bound to a specific*** programming ***language*** and its ***syntax***. More importantly, even non-programmers can have a deeper understanding of the program’s internal mechanisms, as algorithms are simple English instructions.

A further benefit of algorithms is that when we deconstruct a program into smaller chunks (decomposition), debugging becomes more unproblematic as we can ***analyse each segment independently***. In cases, algorithms with well-defined functionalities and precisely stated instructions could eliminate bugs even before they would appear in the code-base in a later phase of the development cycle or worst, in the production code.

**Determine the steps taken in writing code.**

When designing new applications, or new functionalities for existing ones, the best-practice is to analyse and define the given task accurately with its input, output, and specific requirements for overall layout and behaviour first, then create algorithms with flow-charts, if necessary. Our ***algorithm will serve as a backbone*** of our logical approach. This phase of development intends to compose a ***generalised*** yet precise idea of the flow of the program.

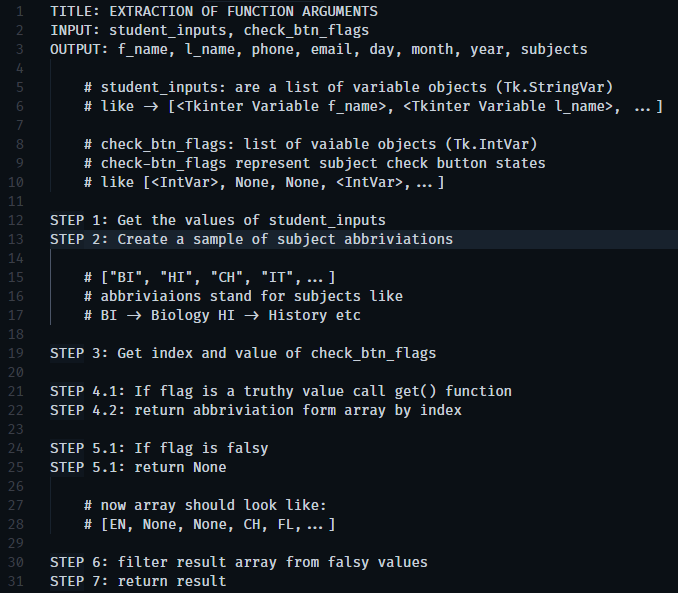
When the logic of the algorithm fulfils all aspects of the requirements, we can transform it into pseudocode. Many developers write algorithms like pseudocode, but these two terms are not interchangeable. Pseudocode is “*a set of practices and conventions for producing very precise, minimally ambiguous descriptions of algorithms.*”(Bennett, 2015) Pseudo-code features various ***flow-control structures***, such as if-else, do-while, switch-case, but it has no special syntax of its own, and the programmer can accommodate it similar to his language preference or adjust it to the company's guide-line.

Pseudo-code is typically written in a text format with the extension of .txt, .doc, therefore it is *not* an executable file, and can occasionally be somewhat confusing for readers with no prior experience in programming. It is an algorithm written with a non-specific syntax, and such, it is situated halfway between algorithms and programming languages. As creating algorithms and pseudo-code can be time-consuming, some developers write code ***impulsively*** without any preparatory measures, consequently creating ***malfunctions*** in existing features or even breaking the mechanism of an entire application.

When the algorithm or pseudo-code - ideally both - is completed, then we can finally convert it into the actual source code. This conversion is less challenging for developers than jumping straight into writing the code in specified programming or scripting languages. If a solution to a problem seems overly complicated without a good reason behind, the development team or a programmer might restructure the algorithm's logic or experiment with other approaches. Changing strategies in the initial stages is cost and time-efficient compared to rewriting or debugging thousands of lines of code.

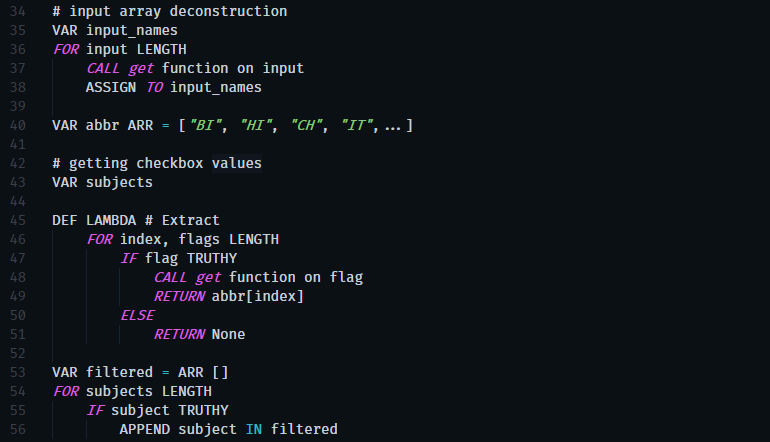
Next, programming languages and frameworks are chosen, tailored to the specifics of the application, as opposed to personal preferences. If the preceding steps were performed in an organised manner, the actual development should go on without much hassle. While generating the source code, every member of the team has to commit their code to a version control system (Git, GitHub), and each change to the master branch must be tested before commit. Repositories can be forked and merged but merging to the master branch is usually done by a senior member of the team. If the application has reached the required standards and it had gone through numerous in-depth functionality and unit testing by QA (Quality Assurance) team, the application is ready for deployment and release, in one or more stages, according to business requirements.

**Examine the implementation of an algorithm in a suitable language. Evaluate the relationship between the written algorithm and the code variant.**



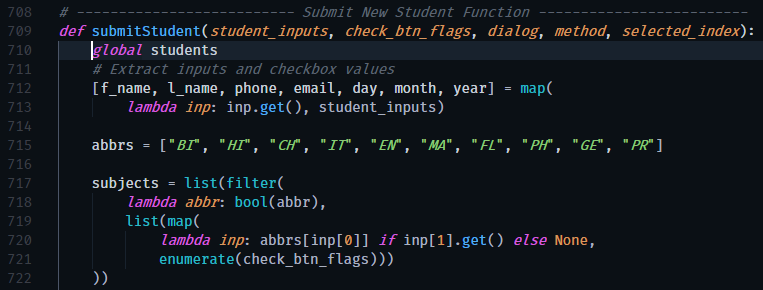
**Figure 1** - Algorithm of Function Argument Extraction

As we can see, the algorithm is defining the cardinal steps of the program in conventional English, but it lacks the specifications of syntax and language context. Even if the above example is an abstract problem, and can be challenging to comprehend for non-programmers completely, it has the potential of ***representing the overall idea*** behind the program.



**Figure 2** – Pseudocode

As we can see (figure 2), our pseudocode does not reflect any specific languages' syntax, and it temporarily obtains several traits from many languages (VAR, DEF, LAMBDA, IF, FOR). Meanwhile, it also gives the freedom for creating fictional language components, such as ARR, CALL.



**Figure 3** - Source Code

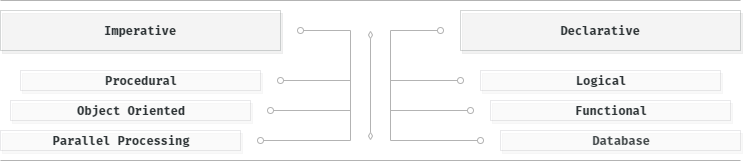
The real implementation (figure 3), however, cannot imitate the pseudocode entirely because of the characteristics of the Python language. Even though the similarities are insignificant at first glance, the applied logic - in all three examples - is identical.

**Programming paradigms**

"By the word paradigm, we understand a set of ***patterns and practices*** used to achieve a certain goal. For an idea to become a paradigm, it should be picked up globally in many independent ***organisations and societies***."(Bartoníček,2014)

There are many programming paradigms in use today:

* ***Imperative:*** uses statements to change a programs state. It is generally ***written linearly***. "Imperative programming developed from consideration of the hardware."(Harle, 1988) A good indicator of the imperative approach is the presence of GOTO statements, like in Basic, Fortran or Assembly.
* ***Object-oriented (OOP):*** As Dr Nurnberg explained it at the Imperial College of London: that OOP encourages to ***decompose*** a problem into related ***subgroups***, where each subgroup becomes a ***self-contained object*** that contains its instructions and data that relate to that object. The four principles of OOP are ***abstraction, encapsulation, inheritance*** and ***polymorphism***. There are many programming languages for OOP, like Java, Python and Ruby, and many others that support classes and OOP design like JavaScript through - so-called - syntactic sugar.
* ***Functional:*** handles computation as of mathematical functions and avoids changing the state of global data. Functional programming uses ***pure functions*** that always return identical results if it is invoked with the same arguments, therefore ***preventing side effects***. Languages built around functional programming are Haskell, Swift, Clojure.
* ***Logical:*** express facts on problems in a system of formal logic. Rules are addressed as ***logical clauses***, including a head and a body. For instance, A is true if B1, B2, and B3 are all true. Such languages are Data Log and Pro Log.
* ***Event-driven****:*is a programming paradigm in where events like a mouse click, keypress or a message from another program determine the progress of the execution of the application*.* An event-driven application is intended to monitor events as they occur, and then invoke an appropriate ***event-handling procedure***. JavaScript, for instance, applies an event-driven model in the browser for fetching ***asynchronous requests***, or handling interaction with the browser.

****

**Figure 4** - Programming Paradigms

**Evaluation of source code in the terms of programming paradigms.**

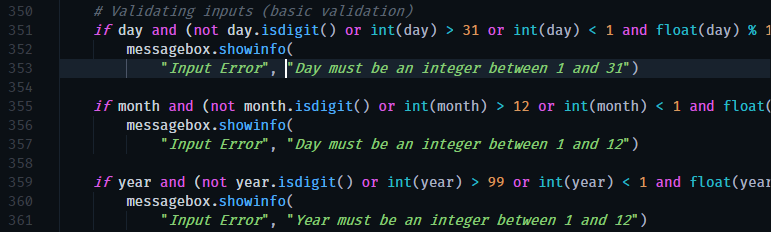
**Imperative:**



**Figure 5** - Imperative Paradigm Example

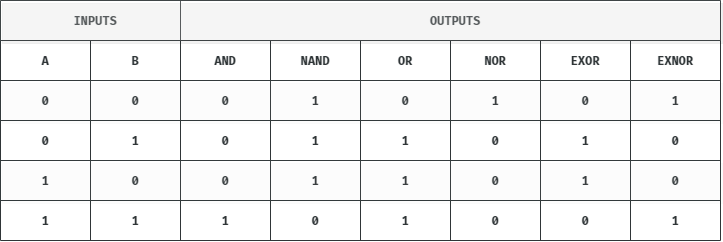
This section of the program linearly changes the graphical state by setting the visual attributes of the components statement by statement.

**Logical:**



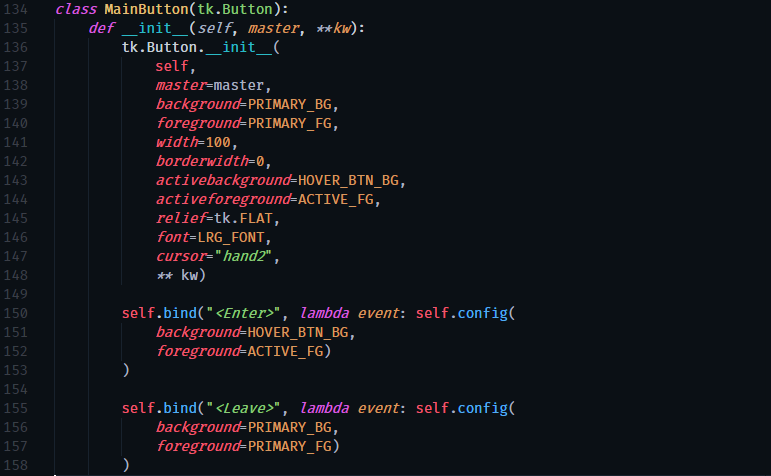
**Figure 6** - Logical Paradigm Example

Input validation gives a fair opportunity for showcasing the application of logical paradigm. Inside of an IF statement, the given condition always boils down into a Boolean value.



**Figure 7** - Logical Inputs and Outputs

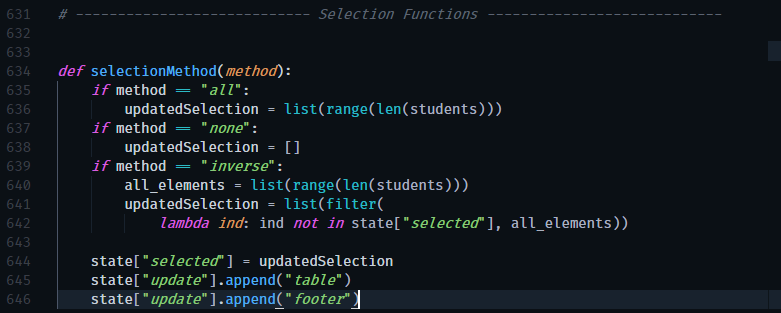
**Object Oriented:**



**Figure 8** - Object Oriented Paradigm Example

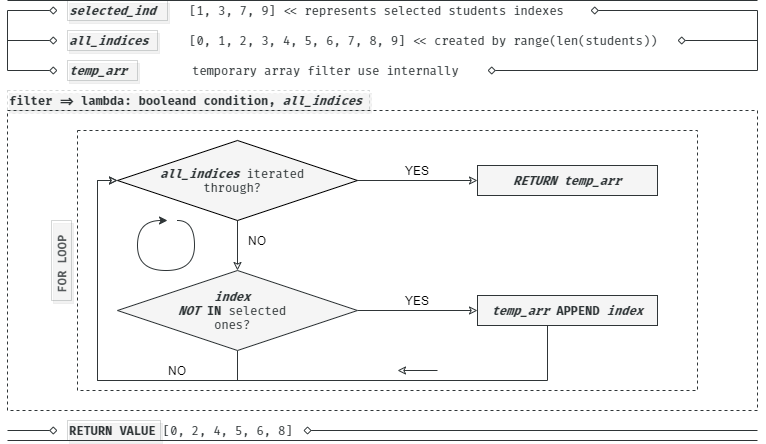
An object-oriented principle, the **inheritance** is used by creating custom buttons for dialogues and main menu. Main button inherits all the necessary functionalities through the graphical user interfaces built-in button. Each time the constructor creates an instance, the parent object will provide its methods to its children, so we can **abstract** existing functionalities rather than building new ones. Although GUI styles could have been used to customise the appearance of the button component, designing a custom button object proved useful when the mouse hover effect was created, as GUI did not provide a built-in solution.

**Functional:**



**Figure 9** - Functional Paradigm Example

When the selection-function is called with its method parameters assigned to ‘*inverse*’, an **anonymous function (lambda)** was used. This lambda function will never directly change the applications state outside of its scope, and when it is called with the same array of numbers, it will return with the same result each time it is called. Therefore, it is a **pure function**, which is a fundamental building block of functional programming.



**Figure 10** - Filter Function Flowchart

**Event-driven:**



**Figure 11** - Event-driven Paradigm Example

Listening-state function periodically ***monitors*** the programs state for changes. Every 100 milliseconds the function is called and reads the state object's update property, which is an array that contains the graphical components’ name that need refreshing, such as the student-list table or the footers pagination. Here further improvement could be to section the footer into segments, like the filter, the pagination or the container of the selected elements and moderate update lags by directly refreshing the affected section rather than the whole footer.

**Features of IDE-s**

Arduino IDE's creators concisely described their product like this: "IDE stands for ***Integrated Development Environment***. It is an official software introduced by Arduino.cc, that is mainly used for ***editing, compiling and uploading*** the code in the Arduino Device." (Fezari, 2018)

Apart from Arduino, there are myriad of IDEs out there, such as Eclipse, PyCharm, IntelliJ, NetBeans and personal favourites like VS Code and Atom. Although we can always write our code in one of the plain text editors, for more complex programs, it is well recommended to use an IDE. The examples below will prove that using an IDE can not only make development a more comfortable process but can also accelerate the speed of programmers’ work.

***Syntax highlighting*** might be considered just a convenient feature, but let us see how quicky would anyone read or make out any sense from the sample lines without knowing the intention or context of the code that were randomly taken from the source code. (*figure 12*)



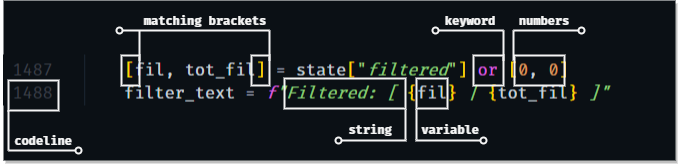
**Figure 12** - Plain Text Code Example

Comparing it to *figure 13*...



**Figure 13** – IDE (VS Code) Code Example

Which example needed less process-time to associate some of the elements with any meaning or purpose? Now, let us discover how we can benefit from the extra information we may extract from these two lines without being familiar with the source-code. (figure 14)



**Figure 14** - Syntax Highlighting Dissected

With a little bit of experience, we can easily spot out typos, as they appear in different colours thanks to the syntax highlighting.

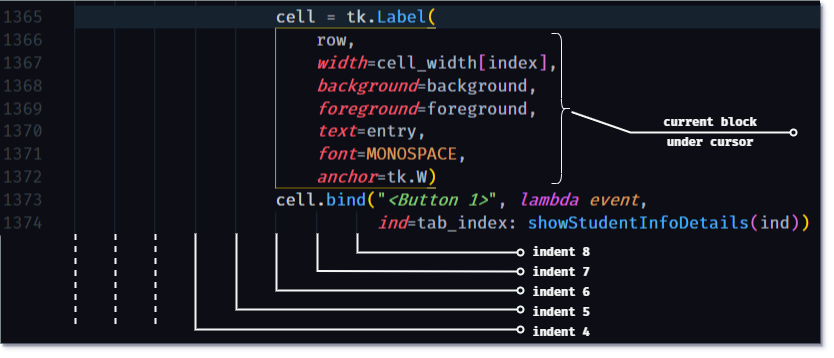
For instance, instead of typing **"*map*"** we accidentally typed **map**. Even before executing the code, it will be apparent that something must be incorrect, as "map" in quotations is a string and, in this instance, would be green, while map as a keyword and its respective colour should be purple.

***Bracket colouring*** is another excellent feature many IDE's have. As an example, let us try to find the matching bracket.



**Figure 15** - Bracket Colouring

Adhering to correct and consistent indentation is a must in Python and a best practice in general. IDEs, like VS Code, **indents** Python code when hitting ***newline***, and provides ***auto-indent*** for many languages. Besides, with some tweaking in the settings, we can set ***line indents***.



**Figure 16** - Indentation Features

One of the most valuable tools that nearly every IDE supply is **autocompletion**. Utilising autocompletion makes development significantly faster by offering **possible keywords**and**variables in scope**, thus reducing the required amount of typing and occasional **mistyping**.



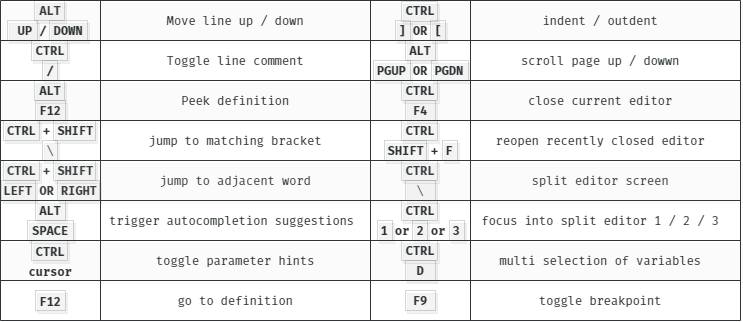
**Figure 17** - Autocompletion

Entering compile and run commands in the terminal is not only a tedious process - that needs to be repeated tremendous amount of times while developing and testing - but is also obsolete. A decent IDE compiles and runs programs, let it be Python, Java or C++. This feature is called **auto compilation**.



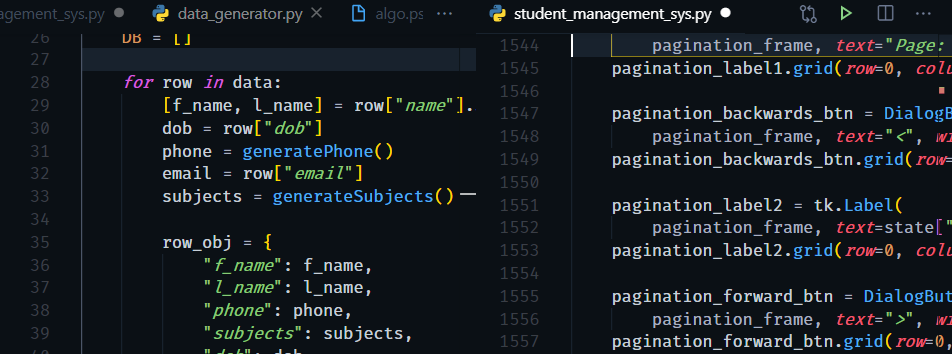
**Figure 18** - Auto compilation

A good code editor beats any general text processor by its useful ***shortcuts*** specialised to programming. Some of the basic ones that programmers use every day are: (figure 19)



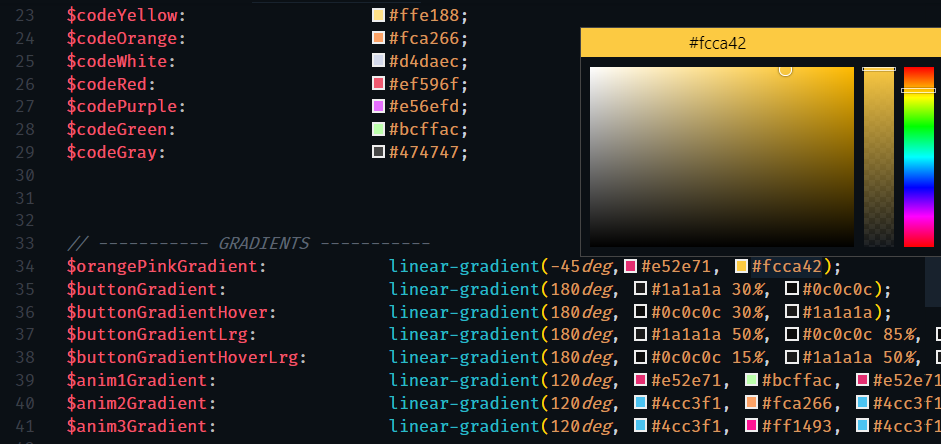
**Figure 19** - Keyboard Shortcuts

For commercial or larger projects, there might be a need for editing on several files simultaneously. In most IDEs, there is an option for ***split-screen***.



**Figure 20** - Split-screen

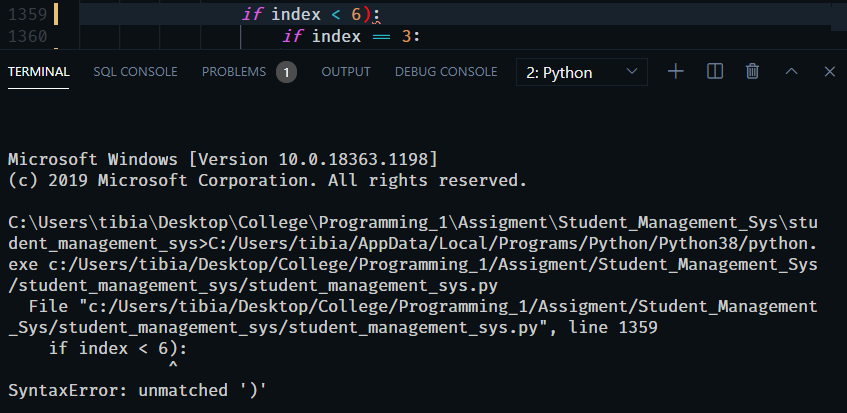
Additionally, some popular extensions amongst front-end developers are ***hot reloading***, that refreshes browser on every save and CSS ***colour indicators***.



**Figure 21** - Built-in Colour Picker and Colour Indicators

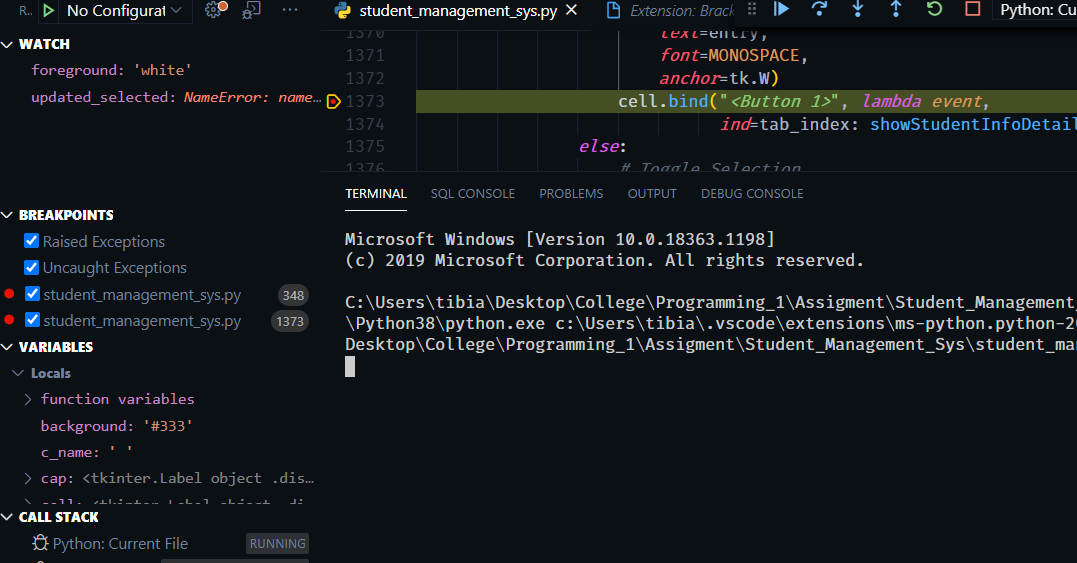
**Testing and debugging with IDEs**

There are many debugging features that IDEs provide, such as **lines under syntax errors**and**error messages**. Simple syntax errors are relatively easy to find. However, many times when an error occurs, the error message might seem ambiguous or does not provide enough information for successful debugging, therefore error messages not only contain the **type of the error** but the exact **program-line** the exception was thrown. More importantly, the message includes the **function** **call stack** or **stack trace error** as well, where we can follow back every function call from top to bottom.



**Figure 22** - Error Message

Many times, the error is not in the syntax of the program, but the logic that it follows. One approach might be brute force **trial and error**, with many lines printed on the **console** in order to find misbehaving variables. When it comes to stubborn and challenging errors, a programmer’s best bet is **watching variables** by setting **breakpoints** and **traceback** any bugs by stepping trough the running program.



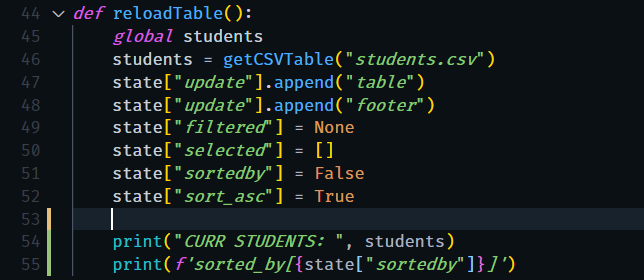
**Figure 23** - VS Code Debugging Tools

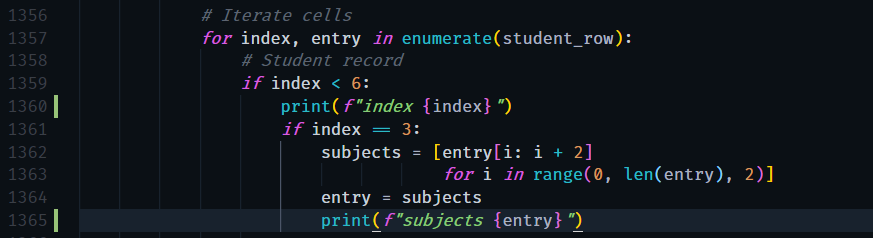
**Importance and methods of testing and debugging process**

Proper testing of the application improves its quality, development time and security, consequently generates **higher profit** and **user trust** by satisfied customers. In industrial development, testing falls on the **QA (Quality Assurance) team**, that scrutinise all possible aspect of the application before release.

There are many types of testing, like:

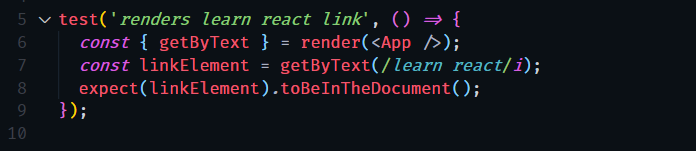
* **unit**-testing: testing whole components or segments,
* **performance**-testing: for reducing overall computation cost,
* **function**-testing: validates function by its parameters and return values,
* **integration**-testing: error-free fitting of components into the application,
* **loops and branch**-testing: checks for the correct execution and variables.

**Figure 25** - Function testing



**Figure 26** - Loop Testing

Apart from debugging in IDEs, several ***testing*** modules can be used for larger volume or commercial applications, such as Jest, Jasmin and Mocha. (Example below has been taken from a different applications source.)



**Figure 24** – Component Testing with Jest

**Development**

Erickson, J., 1999. Algorithms.

Bennett, N., 2015. Introduction to Algorithms and Pseudocode.

Bartoníček, Jan. (2014). Programming Language Paradigms & The Main Principles of Object-Oriented Programming. CRIS - Bulletin of the Centre for Research and Interdisciplinary Study.

Harle, R., 1988. Object Oriented Programming. In *Computer Software and Applications Conference* (Vol. 4, pp. 51-57).

Nurnberg. R. (date unknown) Imperial College of London, Object Oriented Programming in C++.

Fezari, Mohamed & Al Dahoud, Ali. (2018). Integrated Development Environment "IDE" For Arduino.