Evaluating The Learnability of Programming Languages

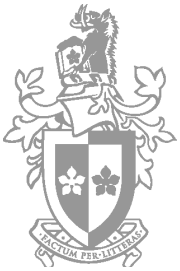
HIT1301 - Algorithmic Problem Solving

Swinburne University of Technology

Alex Cummaudo

1744070

June 19, 2013



Abstract

Learning programming for problem-solving purposes allow for programming concepts to solve problems via a variety of different tools, or programming languages. However, what happens when one familiar tool is replaced with another—if the problem is familiar and easily solvable with one tool, is this the case with another? By observing an intermediate programmer's ability to learn, it was found that if these concepts are well known, adapting them to another language is not difficult. A survey, which measured the participant's confidence in the language afterwards, suggested that most programmers themselves find little difficulty in learning a language, emphasising that languages are not as difficult to learn as some think to be.

Contents

1 Introduction

1.1 Background

Learning a programming language is very much like learning logic. Programming is a task which tells a computer what to do in a *literal* sense. These literal instructions come in a variety of languages, each with their own syntax and keywords, and thus are often easy to misinterpret by programmers who are unfamiliar not with the *concepts* of these instructions, but the ways in which to implement them in the new languages.

This can be problematic. At times, a programmer may be lead down the wrong path—what they think to be the correct method of solving a problem and how a computer may respond to that method may not align. The programmer is blinded by an incorrect solution since they are unable to visualise the data they are working with or the dynamic behaviour required by the solution (Victor, 2012).

They deem their way as the only way of solving a problem, adamant not to deviate from a path which they think is right but is indeed wrong. They misinterpret this stubbornness—ultimately due to a lack of skill in that new language—as the *only* logic to solve the problem.

This was how this research came into fruition; by observation of other students learning a new language, who were stuck following a path which lead to only more bugs. Hence, this isn't a flaw with the keywords or syntax of a language, but instead a problem on how one thinks:

> Programming is a way of thinking, not a rote skill. Learning about "for" loops is not learning to program, any more than learning about pencils is learning to draw
>
> - Victor (2012)

This report aimed at discussing the ways in which people think by analysing the minds of programmers. The scenarios they faced are familiar, adapted from the coursework of *HIT1301 Algorithmic Problem Solving*. However, when a completely new set of tools (programming languages) to work with were introduced, seeing if this familiar problem became foreign (masked by the new language) was investigated.

1.2 Environment

The purpose of this work was to investigate how easy it is to learn a new programming language. It was assumed that:

- a programmer had prior knowledge of basic programming concepts, and
- a programmer had prior knowledge of any other programming language(s) (henceforth referred to as a *base language(s)*).

1.3 Aims and Goals

By first defining what these programming concepts were, the report had two primary questions to answer, which can then be divided further:

1. How well are programming concepts implemented when the language varies?
   (a) Which concept(s) require the most difficulty to understand?
   (b) Does knowing a *base language* hinder or improve learning?
2. How well do programmers learn a new language?
   (a) How quickly does it take for a programmer to grasp the basics of a new language?
   (b) Does providing visual cues help to learn a new language?

These aims revealed what happens to programmers when the syntactical elements of a programming language change. The amount of difficulty incurred to implement the structure of programming concepts in a new language, with the use of flowcharts and *base language*s, was also assessed.

1.4 Research Methodology

This research involves investigating the ways programmers solve problems in an unfamiliar language. To determine answers to the research questions proposed in the previous section, an variety of data gathering techniques are to be utilised.

1.4.1 Data Gathering Techniques

Good data gathering techniques are reflected in terms of the qualitative characteristics and quantitative figures of the results it displays. Not all research techniques, however, provide results which cover both forms of data. Yoshikawa et al. (2008) note that one form of data and thus one technique alone is an "overly simple conceptualization"—thus by combining two or more investigative techniques the strengths of one technique is emphasised, while its weaknesses are compensated with the strengths of another.

Three possible investigatory techniques that were considered for this study, assessing each of their positives and negatives. They are summarised below in Table 1.

Table 1: A summary of all investigatory techniques considered.

Technique	**Strengths**	**Weaknesses**
Surveys	Surveys are efficient—large amounts of easily-interpretable quantitative data are collected by them. It is also easy to make and distribute surveys.	Surveys are inflexible—as a closed style of questioning, it will not give rise to unexpected answers. Hence, less unexpected qualitative data is measured.
Observations	Observations are flexible—participant's problem-solving methods directly recorded using observational techniques and therefore unexpected methods can easily captured.	Observations are inefficient—good observations take time, and are quite time consuming.
Interviews	Interviews are flexible—participants asked follow-up questions from unexpected answers, collecting wide scope of qualitative data.	Interviews are inefficient—they are time consuming, are subject to the social skills of a participant, and interviewing skills required to acquire good data.

All three techniques were utilised to an extent: surveys allowed for a relatively efficient way to assess a participant's programming background and future confidence in the introduced language. To deal with the inflexible nature of surveys, some questions were purposely

opinion-based, so that unexpected answers were captured. Two surveys were utilised:

1. A pre-evaluation survey measuring:
    - basic demographic information (age, gender etc.),
    - prior experience with programming, and
    - knowledge of programming concepts.
2. A post-evaluation survey measuring confidence in the new language learnt.

Observation of the participant coding as they learn immediately captures difficulties. Short questions will be asked during the observation (as an informal interview) to determine the reasoning behind their problem-solving methods in real-time.

1.4.2 Selection of Introduced Language

Participants were asked to utilise the programming language of *Python* for this study. *Python* is a free, powerful and portable language, used in a variety of domains for projects both large and small. It is also remarkably easy to learn, with the a core language that is simple to beginners and can take only a few days to grasp (Lutz, 2007).

As opposed to other comparable languages such as *Java*, *VB.NET*, *Ruby* or *C++*, *Python* was a more suitable candidate since, according to Lutz (2007):

- it has a relatively clean syntax over alternatives such as than *Ruby* and *Java* (improving readability and lessening chances of bugs),
- unlike *VB.NET*, it is open source and cross platform (allowing for greater flexibility in the study), and
- it is considered to be simpler and easier to use than *C++*.

1.4.3 Selection of Concepts

Five primary programming concepts of intermediate difficulty were involved in the problems that the participants solved. They are listed below by increasing difficulty:

1. Variables and Expressions

- Declaration
- Assignment
- Types
- Parameters

2. Selectional Programming
    - `If` statements
3. Repetitional Programming
    - `While` (pre-test) loops
    - `Repeat/Do...While` (post-test) loops
4. Functional Decomposition
    - Working with functions
    - Working with procedures
5. Arrays
    - Declaration
    - Assignment
    - Iteration (via `For` loops)

The following definitions declare what is meant by concept and language complexity when referenced further in this report.

Defining Programming Concept Complexity: The study involves a selection of ten problems that gradually introduce more complex programming concepts (i.e., two tasks per concept—see Appendix A). As new tasks introduce increasingly complicated concepts, the complexity of the logic behind each problem is therefore linear (see Figure 1).

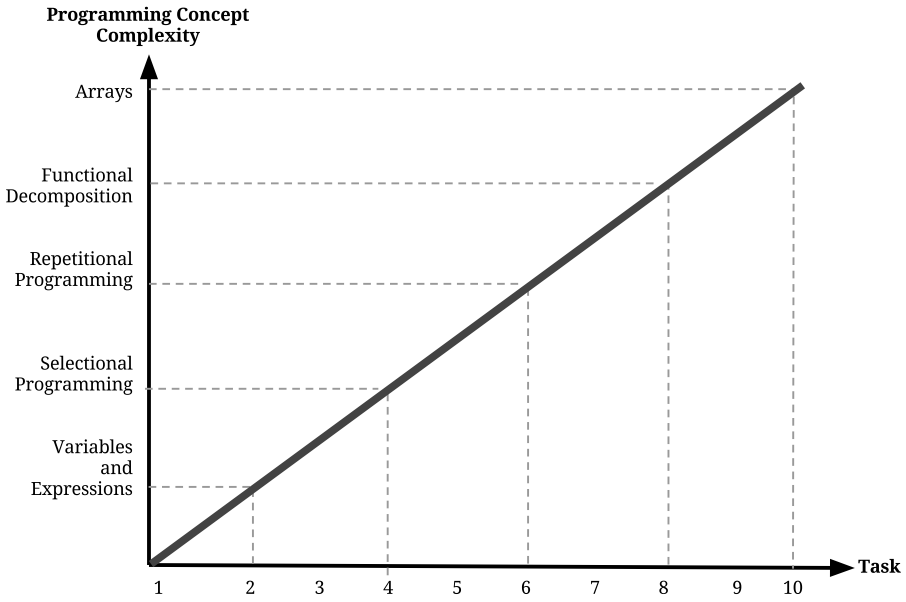


Figure 1: For each progressing task, the complexity of the concept increases

Defining Programming Language Complexity: Ideally, a programmer—with good knowledge of these concepts—should be able to map concepts to a new language. Deviation from the linear relationship in Figure 1 emphasises which concepts are difficult to grasp because of the new syntactical elements in *Python*, not because of the concepts themselves.

Analysing the quality of implementation of each concept (discussed in Section 1.4.5), ultimately highlights the participant's learning of the language (for example, the participant made declaring a variable harder than it needed to be because they didn't know how to do so in *Python*). Deviation from the learning curve is discussed later in the report.

1.4.4 Selection of Participants

This study involved evaluating intermediately experienced programmers. It was required that participants were familiar with the programming concepts outlined in the previous section. To suit the scope of the evaluation, participants were required to know what these programming concepts were and how to utilise them in a problem solving situation. Similarly, the study avoided expert programmers who are quick learners in new languages—a fine balance was achieved by use of the pre-evaluation survey which measured this background.

Participants also must have learnt at least one *base language*—that is *Pascal*. This is because *Pascal* is a suitable learning language, allowing first-time programmers to "avoid getting bogged down in the details of the syntax of a programming language" (Bell et al., 1987). *C* was also a secondary *base language*, since it is a widely adopted language and due to its ubiquitous nature—especially in terms of its derivatives *C++* and *C#*, both of which are widely adopted as well (Griffiths, 2012).

These two were appropriate languages for participants who were studying *HIT1301 Algorithmic Problem Solving*, who were required to study both *Pascal* and *C* over the first, 2013 semester.

1.4.5 Measuring Participant Data

As discussed in Section 1.4.4, a pre-evaluation survey was utilised to determine which participants were most suited to the scope of the study. The post-evaluation survey measured how confident the participant was in their newly learnt language.

During the observation, specific data was recorded for further analysis. Table 2 describes several metrics that were implemented and their usefulness to the study.

By recording the terminal output of each participant's session, the output and learning curve of the programmer was then assessed, based on whether or not their solutions were valid.

Each compiler issue was recorded, categorised into either a problem with poor implementation of a programming concept or poor use of *Python* syntax or elements (determining whether the participant deviated from the complexity curve as discussed on Page 8 via Figure 1). Participants, when needed, were allowed hints from the instructor for each task if they found that problem difficult to solve. The assist given was then recorded as well as what the assist was for (e.g. a programming concept or language problem.) This too helped define any deviation from the complexity curve.

The speed of writing a working solution without any errors was also recorded, to indicate the average time a programming concept in *Python* syntax was learnt.

Participants were not forced to complete every task (as outlined in Section 2.2.1) and were free to skip a task if they found the problem too difficult to solve. Since every problem related to a specific programming concept, this too emphasised which programming concepts were difficult to implement. Where a task was completed, the number of lines written were recorded, since unnecessary code is usually proportional to unfamiliarity of the language

Table 2: A summary of metrics to be used in this study, and the use they provide.

Metric	**Usefulness**
Number and type of bugs when compiling	This measured structural and syntactical difficulty when participants compiled their programs. Analysis indicated if they found errors confusing.
Time taken to write a working program	This measured speed of learning of participants.
Number of lines of code	To measure complexity and confusion over certain problems, programming concepts, or *Python* elements, lines of code were recorded. When unnecessarily long, this indicated that the participant did not know what they were doing, expressing this by writing more than what was needed.
Number and types of assists from instructor	This measured confusion regarding certain problems, programming concepts, or *Python* elements. The types of assists was qualitative data, which gives greater insight than just quantitative figures.
Comments from participants during evaluation	Participants gave qualitative information regarding certain problems, programming concepts, or *Python* elements. They were encouraged to give as much qualitative information as possible, or were asked questions regarding their problem-solving methods.

(Lipow, 1982), indicative of the progress of participant's learning.

Any questions and responses from the instructor / participant during the observation were recorded. These questions were based on the methods used by participants to solve the tasks asked of them.

2 Method

2.1 Required Materials

The observation was recorded in that all code and terminal output was captured (see Appendix B.) Time was also measured. Some form of recording equipment was hence needed. Other observations, such as assists needed by participants, were recorded by the investigator with a pen and paper.

The following materials were required to carry out the study:

- Laptop with a UNIX shell,
- *Python* interpreter,
- Timer
- Text editor, where participants have a selection from:
  - Sublime Text 2 (OS X environment), or
  - Notepad++ (Windows environment)
- Pen and paper.

A visual flow of each problem (and how it could be solved) was provided, by using one or more of:

- a flowchart,
- a compiled example of a solution,
- an uncompiled example of a working solution written in a *base language*.

These diagrams and code were provided in Appendix A.

A guide on the syntax and elements were also developed for the participant's use—based on the guides of Lambert and Osborne (2012) and Beazley (2009)—which is attached in Appendix A.

2.2 Procedure

2.2.1 Ethical Treatment

Treating participants ethically is achieved by using informed consent forms (see Appendix A), which outline their rights: that their data will be anonymised, that consent forms and data are detached and who viewers of their data will be. This ensures privacy is maintained.

To maintain the participant's freedom, the form outlines their right to withdraw at any time from any questions or tasks asked of them for any reason.

2.2.2 Process

After finding a suitable time and location for each participant, the pre-evaluation survey was distributed. The participant is then given a task sheet (see Appendix A) with ten problems to solve by using *Python*, as well as a guide on how to write in *Python*. After this is completed, the post-evaluation survey (see Appendix C) is distributed. The overall task took thirty minutes to an hour depending on the participant's learning curve. An in-depth, chronological procedure of the investigation is as follows:

1. Contact participants. Organise time and location to conduct the evaluation that suits them.
2. Set up a laptop with the appropriate applications (see section 2.1).
3. Introduce the purpose of the study and what it is that will be evaluated on them (be clear that it is not to evaluate how well they *code*, rather how they *learn*).
4. Introduce the informed consent form (see Appendix A). Give time to read it. Ask to sign it.
5. Introduce the pre-evaluation survey. Allow time to complete it.
6. Introduce the *Python* guide, and then the task sheets.
7. Begin the process of coding;
    - begin timing each task
    - capture terminal output

- ask the participant questions (if needed) about their methods

8. Introduce the post-evaluation survey
9. Once completed, thank the participant for their time.

3 Results

The source of the following data can be found in Appendix C

3.1 Participant Survey Data

3.1.1 Pre-Evaluation

Three participants were evaluated for this study. Participants were all male, in an age bracket of 18 to 34. As such, there are limitations of having such a small sample size. This limitation is discussed in greater detail in Section 4.4, Page 26.

All participants had prior programming experience—in fact, the majority of participants described that they first programmed at least three to five years ago. Participants answered that they were intermediately experienced with programming concepts, and when asked to rank their ability on a scale of one to four (four being an expert) the average was 3. This figure was as expected; participants were not experts, but *intermediate*—perfect for the scope of this study (see Section 1.4.4.)

Most participants were familiar with the two *base languages* described in Section 1.4.2, although *VB.NET* and *JavaScript* were also a popular choice in languages prior learnt (see Figure 2). However, two participants had some prior experience with the *Python* programming languages beforehand, which affected the results (see Section 4.4.)

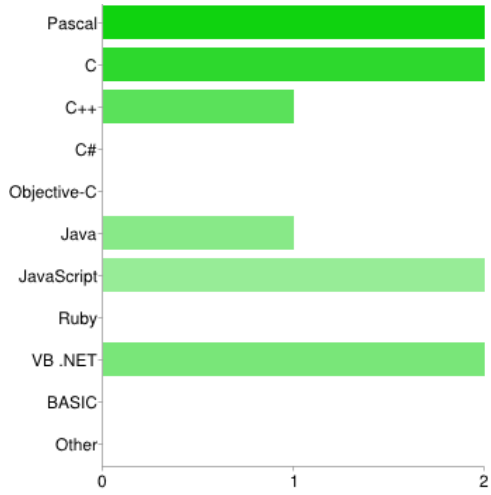


Figure 2: Responses to the pre-evaluation question:
Other than Python, what coding languages do you know or have attempted to learn?

Table 3: Summary of participants poor responses to programming concept understanding

Concept	Comments
Expressions	Confused expression for equation. Poor understanding of boolean expressions.
Types	Believed boolean can have one value only—possible chance of misinterpretation of question
Variables	Unclear of parameter definition
Repetition	Confused `Do...While` loop for a pre-test loop
Functions and Procedures	Unclear of advantages of functions over procedures.
Arrays	Poor understanding that array is a single variable

Participants were largely familiar with basic programming concepts, which Table 3 summarises. Main concerns arose in for functional decomposition in that there was poor understanding of a parameter—only one participant could correctly define a parameter. Other errors in defining programming concepts came from a minority of participants. Thus, it is clear that the participants assessed were intermediate in their knowledge of programming concepts—they were neither experts in knowing basic programming concepts nor were they novices.

From the pre-evaluation survey, participants were also familiar with how to read flowcharts, and thus were able to utilise them in the tasks.

3.1.2 Post-Evaluation

After the evaluation was conducted, participants generally found that *Python* is an "efficient" and "easy" language to work with, likely due to it's significant use of English-based keywords.

As expected, the most difficult concept to implement in *Python* was arrays, while variables and expressions were easier to implement. This is in-line with the complexity curve outlined in Section 1.4.3, Figure 1—arrays were expected to be harder to implement since they are

different to reference and define in *Python*, as opposed to *Pascal* or *C*. This was a common source of errors made by participants and is discussed further in Section 3.2.1.

Ultimately, the majority of participants were largely confident in continuing their *Python* learning. On a scale of one to four, with four being very confident, 67 per cent of participants gave a score of four. This confidence was fortified from the *Pascal* source code and flowcharts, which—as intended—allowed participants to visually compare a familiar problem in an unfamiliar language.

3.2 Evaluation Data

3.2.1 Bugs Encountered

Bug count increased in a positive fashion (see Figure 3.) This was particularly clear towards the later-half of the evaluation where participant's faced problems that required more complicated syntax that was somewhat different to that of a *base language*.

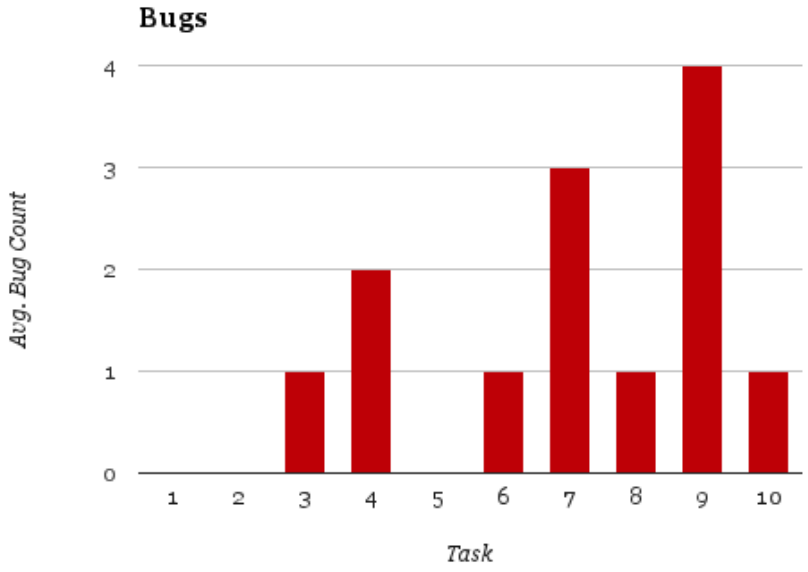


Figure 3: Average number of bugs encountered by participants

Tasks which made use of `for` loops and arrays proved particularly challenging to participants. From Figure 3, the greatest spike occurs at task nine which introduced arrays and `for` loops. Unlike *C* or *Pascal*, variables in *Python* must be defined *and* initialised on the same line of code. Participants understood this concept well for all other kinds of variables, such as strings or integers, but struggled to grasp how to do this with arrays; they seemed to dissociate arrays from being just another kind of variable. This dissociation caused participants to declare an array in a different manner (e.g., `foo[10]` to define an array of 10 indexes). This often cause confusion and bugs.

When asked about their thoughts on arrays and `for` loops, participants noted that arrays "frustrate me" and that *Python* has "a very different for loop", thereby highlighting that this was the most difficult concept to handle. This is further fortified from the post-evaluation

survey responses, as discussed in Section 3.1.2.

Peaks of bugs at tasks four and seven were primarily due to syntactical errors of boolean conditions. Often, a semicolon was not at the end of each condition, or a comparison equals (`==`) was misused with an assignment equals (`=`). These errors were mainly due to differences in syntax between a *base language* and *Python*. However, by task eight, where boolean conditions were also used, these bugs were infrequently encountered, emphasising that participants had become used to this new syntax.

The following observations were frequently common throughout the evaluation:

- Participants failed to understand that a variable does not require an type when declared, but a value.
- Participants improperly used the `int` function:
  - It took several bugs for participants to realise that the `int` function is only needed to convert numeric terminal input (i.e., a string) into an integer for arithmetic purposes.
  - Some participants declared an integer variable using the `int` function—i.e., they used `foo = int(100)` rather than `foo = 100`.
- Participants failed to include a colon to declare a new block of statements.
- Participants misused the equals symbol in a condition (used assignment equals, `=`, rather than comparison equals, `==`)
- Participants manually declared a control variable for a `for` loop.
- Participants improperly referenced an array's index value within an `for` loop (used `foo[i]` rather than just `i`.)

Assists were often required for figuring out syntactical differences between the *Pascal* source code supplied and how to 'convert' that into *Python*. This was expected, since the primary sources of bugs were due to invalid syntax occurring (see Table 4). However, on average, few assists were needed by participants over all of the tasks and each participant usually only required just one over their evaluation.

Table 4: Common bugs participants encountered during evaluation

Bug Type	**Count**	**Frequency**
String Concatenation	1	4%
Invalid syntax	13	46%
Indentation error	3	11%
Keyboard interrupt (due to bad programming)	1	4%
Undefined variable reference	3	11%
Incorrect type name	1	4%
Conversion to string from integer error	2	7%
Referenced array value by using array name	2	7%
Tried to iterate i in a for loop manually	2	7%

3.2.2 Code Quality

On inspection of each participant's code in Appendix B, code was generally well-written. Figure 4 shows an increasing trend, emphasising that as tasks increased in complexity, the code required to solve these tasks also increased. Tasks seven, eight and ten required some use of functional decomposition, directly due to an increased number of functions required. Likewise, the amount of code used increased proportionally—explaining the trend below.

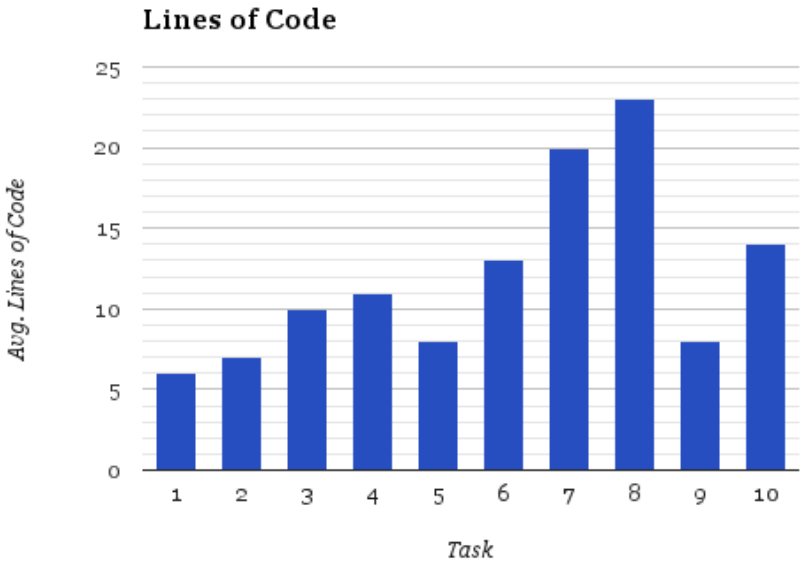


Figure 4: Average number of code lines counted per task

At times, though, use of variables was poorly used. Participant two frequently used global variables for at least half of the tasks asked of them, even though the variable was required in one function only. Moreover, a majority of participants used more variables than were needed:

- For task two, participant two declared a new variable, `newAge`, to add one variable to another (`newAge = age + addAge`) rather than just incrementing `age` by `addAge` (see Page 32 for code.)
- For task six, participant one declared a variable `const` as a constant. They used this to make a constant `True` expression for a boolean condition (`while const == 0 :`) rather than just using a `True` keyword (see Page 38 for code.)

- For task nine, both participant one and two declared a control variable, `i`, and initialised it to zero, even though the `for` loop it was used in did not require an declaration of a control variable. Interestingly, they did not do the same for task ten, which required the same concept to be used (see Page 46 for code.)

3.2.3 Learning Speed

Participants were generally fast learners (as emphasised in Figure 5), and on average took 231 seconds (3 minutes and 51 seconds) to complete tasks. The general incline in time taken is due to increased difficulty with problems.

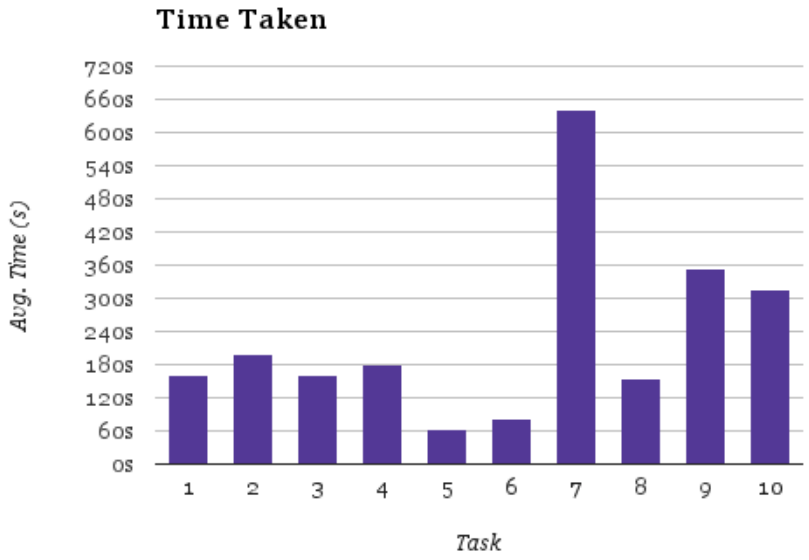


Figure 5: Average time taken for participants to complete each task

Task 7 took particularly longer due to increased difficulty in understanding functional decomposition. A majority of participants were unsure why functional decomposition was required in this problem and did not know how it could be implemented. Moreover, participants were not aware that the flowchart included in Task 7 applied to a *sole* procedure, and not the entire program; participant one mentioned that "these instructions aren't clear. I assumed that I needed to follow the flowchart only".

Thus, poor instructions and poor task description, and not a participant's learning, is to blame for this spike.

4 Discussion

4.1 Quality of Concept Implementation

Intermediate programmers generally implemented programming concepts quite well, even though the language was somewhat unfamiliar to them. Any major sources of errors when doing so are usually due to the technical placement of syntax (see Table 4); syntax placement is therefore the most difficult problem to handle when a new language is learnt. Of the concepts assessed in this study, it is found that programmers find the definition and iteration of arrays to be the most difficult to understand in a foreign language, as outlined in Section 3.2.1.

Functional decomposition in an unfamiliar language is also one that required the most thought from programmers. Participants were not able to visualise how a problem should be broken down in *Python*, and this was often a concept which required the most assistance to understand (see Section 3.2.1.)

Although *flow* of programs (as discussed in Section 4.2) is generally comprehended quite well regardless of the language, *structure* was not as well interpreted; participants often made mistakes in separating code apart from each other—as implied by the 11 per cent frequency of indentation errors in Table 4. A participant therefore had poor understanding of how to implement good structure in *Python*, since indentation is functionally required for a *Python* program to have proper structure (i.e., in the use of code blocks.)

4.2 Visual Cues

Providing visual cues for programmers to visualise how a problem could be solved is helpful for their learning. By providing snippets of the problem in either a *base language's* source code or a flowchart, participants were able to quickly move on with *what* task asked of them and focus on *how* to implement it. However as suggested in Section 3.2.3, participants became too dependent on following a flowchart; they assumed that a flowchart applied for an entire problem and were following the flowchart too explicitly, rather than using it as a visual guide to help solve the problem (i.e., the tasks themselves became too easy to solve when a cue was provided.)

Thus, a heavy reliance on a flowchart or *base language* source code illustrated that programmers do not think about how to solve problems, but think how they can merely 'convert'

the visual aid into a foreign language's source code—which may not necessarily mean that they are *learning* the foreign language.

Although cues allow for a problem to be solved regardless of the language it is written in, it perhaps hinders the *understanding* and *retention* of the code written, ultimately affecting their learning.

4.3 Speed of Learning

Programmers tend to quickly grasp the basics of a new language in under a time period of about forty-five minutes (a value greatly overestimated prior to the evaluation.) On inspection of Figure 5, it is seen that by tasks five and six, participants understood the basics of *Python*.

Hence, when comparing Figures 4 and 5, it takes, on average, a learner 55 lines of code to write fluidly in the foreign language.

4.4 Limitations

4.4.1 Limited Participants

The sample evaluated for the study was very limited. As outlined in Section 3.1.1, participants were all very similar in terms of their demographics. This was due to limited resources; participants were not given any incentives to join in the study and their time was purely voluntary. Participants were also hard to find since the study usually takes about an hour to conduct, and potential participants were usually deterred by this due to their busy workload. This was very disappointing, since it has major impact on the validity of this study; having such a small sample size reduces the results and thus breadth to apply the results to a larger scale is reduced.

Alleviating this for a future study would require a greater selection of intermediately experienced programmers—as many as possible—by increasing resources (e.g., money and time available) and pay potential participants, conducting it in a time of minimal workload (i.e., semester breaks). This would provide for the increased breadth that this study lacks.

4.4.2 Hawthorne Effect

The Hawthorne Effect results in changes in participant's behaviour as they are knowingly being observed (Courage and Baxter, 2005). This hence gives inaccurate results since real-life and non-clinical learning of *Python* would occur outside of a study; participant two stated that they "felt intimidated every time [the instructor] wrote something down". Their distraction could have easily increased the time taken for them to complete that task—they may have second-guessed their programming methods, even if a note taken on them was positive and not negative, and changed something that was initially correct to an incorrect method.

In future, it would be best to make the participant more relaxed before they begin the tasks so that they are habituated, and feel comfortable. This will allow for an environment where their learning is not affected. Perhaps, a field study could be used instead—so that they are already habituated in their own environment—or a better observation system where the participant cannot see an evaluator take notes on them (e.g., analysing what is on their screen remotely via a VNC connection.)

4.4.3 Limited Scope of Participants and Tasks

Two thirds of participants had some previous interaction with *Python*. Hence, bias exists within the participant's evaluated—their previous knowledge of *Python* is not indicative of how well they *learnt* the language, but instead how well they could *remember* it. As a result, averages were brought down by quick participant's who were not learning the language, but simply converting it. This means that the evaluation is not based on the learnability of the language but on the ease of converting it from one language to another, a topic that is in conflict with the aims of this study.

Future studies should make better use of the pre-evaluation survey. Should there be any indication that a participant has interacted with *Python* beforehand, their results should not be considered since results will become bias and not indicative of a programmer's learning capabilities.

5 Conclusion

Programmers are able to adapt programming concepts to new languages generally with ease. It was found that programming concepts, when previously well learnt, are stable in the programmer's mind—those without such a grasp tend to stumble in their learning.

The Complexity Curve (see Figure 1, Page 9) visually demonstrated the *expected* difficulty in learning a language. On inspection of Figures 3, 4 and 5, an increasing, linear trend in a programmer's learning ability aligns with the Complexity Curve's trend, affirming the increasing difficulty of implementing the concepts chosen.

Participants with little exposure to *Python* still grasped the basics of the language in just under an hour, stating that their confidence in the language was strong. Even with relatively familiar problems (e.g. `SumArray` in Task 10) there is little difficulty in adapting a new language to that problem—albeit an initial, steep learning curve (Tasks One to Five) is necessary to attain the basics of the language.

Therefore, programming languages are not difficult to learn since the concepts *behind* each language are *stable*, and—as expected—the difficulty lies only in *attaining* the knowledge of these concepts, not the knowledge in how to *apply* these concepts.

Programmers[1] should have confidence in learning any language they wish to choose. Although most errors lie in just technical syntax, any good programmer should apply their knowledge of programming concepts in a variety of languages.

This achieves greater adaptability in the problems programmers solve: by learning more languages, they have a wider variety of tools to work with, allowing for greater balance of control over what they can do with that language and the language's ability to do more with less code. Programmers therefore have a greater selection of tools to pick one that best suits the problem at hand, yet one that best suits how they think.

[1] These results are not entirely indicative of a real-life scenario, and is in no way applicable to the general 'programmer'. Limited Participants and the Hawthorne Effect both are significant limitations of this study, and therefore incur a lack of generalisation to *most* programmers.

References

D. M. Beazley. *Python essential reference*. Upper Saddle River, NJ : Addison-Wesley, 2009.

D. Bell, P. Scott, and J. Miller. A first course in programming. *ACM SIGCSE Bulletin*, 19 (2):49, 1987. doi: 10.1145/24728.24739.

C. Courage and K. Baxter. *Understanding Your Users: A Practical Guide to User Requirements Methods, Tools, and Techniques*, page 569. San Francisco, Calif. : Morgan Kaufmann, 2005.

D. Griffiths. Why learn c?, 2012. URL `http://programming.oreilly.com/2012/06/why-learn-c.html`. [Online; accessed 11 May 2013].

K. A. Lambert and M. Osborne. *Fundamentals of Python : first programs*, pages 28, 48, 56, 102, 176–178. Boston, Mass. : Course Technology/Cengage Learning, 2012.

M. Lipow. Number of faults per line of code. *Software Engineering, IEEE Transactions on*, SE-8(4):437–439, 1982. ISSN 0098-5589. doi: 10.1109/TSE.1982.235579.

M. Lutz. *Learning Python*, pages 17–18. Sebastopol : O'Reilly Media, Inc., 2007.

B. Victor. Learnable programming, 2012. URL `http://worrydream.com/LearnableProgramming/`. [Online; accessed 6 May 2013].

H. Yoshikawa, T. S. Weisner, A. Kalil, and N. Way. Mixing qualitative and quantitative research in developmental science. *Developmental Psychology*, 44(2):344–354, 2008.

A Supporting Documentation

This Appendix lists the supporting documentation for the study. They are attached in the following order:

1. Statement of Consent Documentation
2. Task Sheet
3. Python Guide

B Participant Output

B.1 Participant Code

Task 1

Participant 1

```
def main():
 name = input("What is your name?")
 print("Hello", name)
main()
```

Participant 2

```

myName = "jryjryjryjryjyrryj"

def main():
 myName = input("What is your name?")
 print("Hello " + myName)


main()
```

Participant 3

```
def main():
 name = input("What is your name?")
 print("Hey", name)

main()
```

Task 2

Participant 1

```
1 def main():
2  age = int(input("How old are you?"))
3  print("In 10 years you will be ", age+10)
4 main()
```

Participant 2

```
1  age = 0
2  addAge = 0
3  newAge = 0
4
5  def main():
6   age = int(input("How old are you? "))
7   addAge = int(input("How many years would you like? "))
8   newAge = age + addAge
9   print("In ",  addAge, " years you'll be ", newAge)
10
11
12 main()
```

Participant 3

```
1 def main():
2  age = int(input("How old are you?"))
3  incrementAge = int(input("How old do you want to be?"))
4  print("In", age, "you'll be", age+incrementAge)
5
6 main()
```

Task 3

Participant 1

```
def main():
 numOne = int(input("First number:"))
 numTwo = int(input("Second number:"))
 if numOne > numTwo :
  print("First number was greater")
 elif numTwo > numOne :
  print("Second number was greater")
 else :
  print("error")
main()
```

Participant 2

```
firstNumb = 0
secNumb = 7
def main():
 firstNumb = int(input("Enter first number "))
 secNumb = int(input("Enter second number "))

 if firstNumb > secNumb:
  print("First value was larger")
 else:
  print("First number was smaller")

main()
```

Participant 3

```
def main():
 firstNumber = int(input("Enter No 1: "))
 secondNumber = int(input("Enter No 2: "))
```

```
4  if firstNumber > secondNumber:
5   print("No 1 was larger")
6  else:
7   print("No 2 was larger")
8
9 main()
```

Task 4

Participant 1

```
def main():
 name = input("What is your name?")
 if name == "Fred" :
  print("Your name is amazing")
 elif name == "Barry" :
  print("nobody's called Barry")
 else :
  print("You have a dumb name")
main()
```

Participant 2

```
name = ""

def main():
 name = input("What is your name? ")

 if ((name == "Fred") or (name == "Frank")):
  print(name, " is a great name!")
 elif name == "Jerry":
  print("I don't like that name.")
 else:
  print("Go away")

main()
```

Participant 3

```
def main():
 name = input("Enter your name: ")
 if name == "Fred" or name == "Frank":
```

```
4    print(name, "is a name I like!")
5   elif name =="Jerry":
6    print("I don't like that name")
7   else:
8    print("Go away!")
9
10 main()
```

Task 5

Participant 1

```
def main():
 i = int(1)
 while i <= 10 :
  print(i)
  i+=1
main()
```

Participant 2

```
def main():

 i = 0

 while i <= 10:
  print(i)
  i += 1


main()
```

Participant 3

```
def main():
 i = 0
 while (i <= 10):
  print(i)
  i += 1

main()
```

Task 6

Participant 1

```
def main():
 const = int(0)
 i = int(10)
 while const == 0 :
  print(i)
  i-=1
  if i <= 0 :
   break;
main()
```

Participant 2

```
def main():

 i = 10
 while True:
  print(i)
  i -= 1
  if i < 0:
   break


main()
```

Participant 3

```
def main():
 i = 10
 while True:
  print(i)
  i -= 1
```

```
6   if i <= 0:
7    break
8
9 main()
```

Task 7

Participant 1

```
1  import random
2
3  def checkGuess(randomNum,guessNum):
4   if guessNum == randomNum :
5    print("You got it!")
6   elif guessNum > randomNum :
7    print("Try a lower number")
8   elif guessNum < randomNum :
9    print("Try a higher number")
10  else :
11   print("error")
12
13 def main():
14  randomNum = random.randint(1,10)
15  while True :
16   print("Enter a number between 1 and 10")
17   guessNum = int(input("Your number:"))
18   checkGuess(randomNum,guessNum)
19   if randomNum == checkGuess :
20    break;
21 main()
```

Participant 2

```
1  import random
2
3  def checkGuess(guessNo, randomNumber):
4   if guessNo > randomNumber:
5    print("too high!")
6   elif guessNo < randomNumber:
```

```
  print("Too low")

def main():
 randomNumber = random.randint(1,10)
 while True:
  guessNo =  int(input("Enter value between 1 and 10: "))
  checkGuess(guessNo, randomNumber)
  if guessNo == randomNumber:
   break

 print("Wow you got it! It was ", guessNo)


main()
```

Participant 3

```
import random

def checkGuess(guessNo, randNo):
 if guessNo == randNo:
  print("Wow, you got it. It was ", randNo)
 elif guessNo > randNo:
  print("Too high")
 elif guessNo < randNo:
  print("Too low")

def main():
 randomNo = random.randint(1,10)
 while True:
  guessNo = int(input("Enter a guess: "))
  checkGuess(guessNo, randomNo)
  if guessNo == randomNo:
   break
```

```
18
19 main()
```

Task 8

Participant 1

```
1  import random
2
3  def checkGuess(randomNum,guessNum):
4   if guessNum == randomNum :
5    print("You got it!")
6    return True
7   elif guessNum > randomNum :
8    print("Try a lower number")
9    return False
10  elif guessNum < randomNum :
11   print("Try a higher number")
12   return False
13  else :
14   print("error")
15   return True
16
17 def main():
18  randomNum = random.randint(1,10)
19  while True :
20   print("Enter a number between 1 and 10")
21   guessNum = int(input("Your number:"))
22   if checkGuess(randomNum,guessNum) == True :
23    break;
24 main()
```

Participant 2

```
1  import random
2
3  def checkGuess(guessNo, randomNumber):
```

```
 if guessNo == randomNumber:
  print("Wow you got it! It was ", guessNo)
  return True
 if guessNo > randomNumber:
  print("too high!")
  return False
 elif guessNo < randomNumber:
  print("Too low")
  return False

def main():
 randomNumber = random.randint(1,10)
 while True:
  guessNo =  int(input("Enter value between 1 and 10: "))
  #checkGuess(guessNo, randomNumber)
  if checkGuess(guessNo, randomNumber) == True:
   break




main()
```

Participant 3

```
import random

def checkGuess(guessNo, randNo):
 if guessNo == randNo:
  print("Wow, you got it. It was ", randNo)
  return True
 elif guessNo > randNo:
  print("Too high")
```

```
9    return False
10  elif guessNo < randNo:
11   print("Too low")
12   return False
13
14 def main():
15  randomNo = random.randint(1,10)
16  while True:
17   guessNo = int(input("Enter a guess: "))
18   if checkGuess(guessNo, randomNo):
19    break
20
21 main()
```

Task 9

Participant 1

```
import random

def main():
 Rainbow = ["Red","Orange","Yellow","Green","Blue","Indigo","Violet"]
 i = int(0)
 for i in Rainbow :
  print(i," is in a rainbow")

main()
```

Participant 2

```
def main():
 Rainbow = ["Red", "Orange", "Yellow", "Green", "Blue", "Indigo", "Violet"]
 i = 0
 for i in Rainbow:
  print(i)

main()
```

Participant 3

```
def main():
 rainbow = ["Red", "Orange", "Yellow", "Green", "Blue", "Indigo", "Violet"]

 for (i) in rainbow:
  print (i)

main()
```

Task 10

Participant 1

```
1  import random
2
3  def sumArray(numbers):
4   runningTotal = 0
5   for i in numbers :
6    runningTotal = i+runningTotal
7   return runningTotal
8
9
10 def main():
11  numbers = [5,10,15]
12  print(sumArray(numbers))
13
14 main()
```

Participant 2

```
1
2
3  def SumArray(array):
4   total = 0
5
6   for num in array:
7    total += num
8   return total
9
10 def main():
11  numbersToAdd = [5, 10, 15]
12  print("The total is ", SumArray(numbersToAdd))
13
```

```

main()
```

Participant 3

```
def sumArray(data):
 runningTotal = 0
 for (i) in data:
  runningTotal += i
 return runningTotal


def main():
 data = [5,10,15]
 print(sumArray(data))

main()
```

B.2 Participant Terminal Output

Task 1

Participant 1

```
1 ===========
2 DATE: 2013-05-17
3 TIME: 13:58:29
4 ===========++++++++++++++++++++++Success!
5 What is your name?aecd
6 Hello aecd
```

Participant 2

```
1 ===========
2 DATE: 2013-05-17
3 TIME: 15:48:20
4 ===========
5 ++++++++++++++++++++Success!
6 What is your name?Fe=red'
7 Hello Fe=red'
```

Participant 3

```
1 ===========
2 DATE: 2013-05-20
3 TIME: 18:55:37
4 ===========++++++++++++++++++++Success!
5 What is your name? Marc
6 Hey Marc
```

Task 2

Participant 1

```
1  ===========
2  DATE: 2013-05-17
3  TIME: 14:01:39
4  ===========
5  How old are you?21
6  Traceback (most recent call last):
7    File "/Users/Alex/dropbox/study/p1/t2.py", line 4, in <module>
8      main()
9    File "/Users/Alex/dropbox/study/p1/t2.py", line 3, in main
10     print("In 10 years you will be ", age+10)
11 How old are you?
12 ===========
13 DATE: 2013-05-17
14 TIME: 14:02:04
15 ===========++++++++++++++++++++++Success!
16 How old are you?32
17 In 10 years you will be  42
```

Participant 2

```
1  ===========
2  DATE: 2013-05-17
3  TIME: 15:54:07
4  ===========
5  SyntaxError: invalid syntax
6    File "/Users/Alex/Dropbox/study/p2/t2.py", line 7
7      print("In " + addAge + " years you'll be " age + addAge)
8                                                   ^
9  ===========
10 DATE: 2013-05-17
```

```
TIME: 15:54:34
==========
How old are you?7
How many years would you like?4
Traceback (most recent call last):
  File "/Users/Alex/Dropbox/study/p2/t2.py", line 10, in <module>
    main()
  File "/Users/Alex/Dropbox/study/p2/t2.py", line 7, in main
    print("In " + addAge + " years you'll be " + age + addAge)
TypeError: Can't convert 'int' object to str implicitly
==========
DATE: 2013-05-17
TIME: 15:55:24
==========
How old are you? 4
How many years would you like? 5
Traceback (most recent call last):
  File "/Users/Alex/Dropbox/study/p2/t2.py", line 12, in <module>
    main()
  File "/Users/Alex/Dropbox/study/p2/t2.py", line 9, in main
    print("In " + addAge + " years you'll be " + newAge)
TypeError: Can't convert 'int' object to str implicitly
==========
DATE: 2013-05-17
TIME: 15:56:43
==========++++++++++++++++++++++Success!
How old are you? 4
How many years would you like? 5
In  5  years you'll be  9
```

Participant 3

```
==========
DATE: 2013-05-20
```

```
3 TIME: 19:05:48
4 ==========++++++++++++++++++++++Success!
5 How old are you?100
6 How old do you want to be?20
7 In 100 you'll be 120
```

Task 3

Participant 1

```
1  ===========
2  DATE: 2013-05-17
3  TIME: 14:05:14
4  ===========
5    File "/Users/Alex/dropbox/study/p1/t3.py", line 3
6      numTwo = int(input(Second number:"))
7                                         ^
8  SyntaxError: invalid syntax
9  ===========
10 DATE: 2013-05-17
11 TIME: 14:05:28
12 ===========++++++++++++++++++++++Success!
13 First number:2
14 Second number:5
15 Second number was greater
16 ===========
17 DATE: 2013-05-17
18 TIME: 14:05:36
19 ===========++++++++++++++++++++++Success!
20 First number:100
21 Second number:2
22 First number was greater
```

Participant 2

```
1  ===========
2  DATE: 2013-05-17
3  TIME: 16:01:22
4  ===========
5  ++++++++++++++++++++++Success!
```

```
Enter first number 5
Enter second number 8
First number was smaller
```

Participant 3

```
===========
DATE: 2013-05-20
TIME: 19:08:22
===========
  File "/Users/Marc/Desktop/p3/t3.py", line 6
    else
       ^
SyntaxError: invalid syntax
===========
DATE: 2013-05-20
TIME: 19:08:29
===========++++++++++++++++++++++Success!
Enter No 1: 2
Enter No 2: 3
No 2 was larger
===========
DATE: 2013-05-20
TIME: 19:08:33
===========++++++++++++++++++++++Success!
Enter No 1: 4
Enter No 2: 1
No 1 was larger
```

Task 4

Participant 1

```
1  ===========
2  DATE: 2013-05-17
3  TIME: 14:08:17
4  ===========
5    File "/Users/Alex/dropbox/study/p1/t4.py", line 7
6      else print("Your have a dumb name")
7                ^
8  SyntaxError: invalid syntax
9  ===========
10 DATE: 2013-05-17
11 TIME: 14:08:32
12 ===========
13   File "/Users/Alex/dropbox/study/p1/t4.py", line 7
14     else
15         ^
16 SyntaxError: invalid syntax
17 ===========
18 DATE: 2013-05-17
19 TIME: 14:08:46
20 ===========++++++++++++++++++++++Success!
21 What is your name?Pho
22 Your have a dumb name
23 ===========
24 DATE: 2013-05-17
25 TIME: 14:09:02
26 ===========++++++++++++++++++++++Success!
27 What is your name?Barry
28 nobody's called Barry
29 ===========
30 DATE: 2013-05-17
```

```
TIME: 14:09:17
==========++++++++++++++++++++++Success!
What is your name?Fred
Your name is amazing
```

Participant 2

```
===========
DATE: 2013-05-17
TIME: 16:07:34
===========
  File "/Users/Alex/Dropbox/study/p2/t4.py", line 6
    if ((name == "Fred") || (name == "Frank")):
                          ^
SyntaxError: invalid syntax
  File "/Users/Alex/Dropbox/study/p2/t4.py", line 6
    if ((name == "Fred") || (name == "Frank")):
                          ^
SyntaxError: invalid syntax
===========
DATE: 2013-05-17
TIME: 16:07:50
===========
  File "/Users/Alex/Dropbox/study/p2/t4.py", line 8
    elif name = "Jerry":
              ^
SyntaxError: invalid syntax
  File "/Users/Alex/Dropbox/study/p2/t4.py", line 8
    elif name = "Jerry":
              ^
SyntaxError: invalid syntax
===========
DATE: 2013-05-17
TIME: 16:08:02
```

```
==========++++++++++++++++++++++Success!
What is your name? afadsj
Go away
===========
DATE: 2013-05-17
TIME: 16:08:08
==========++++++++++++++++++++++Success!
What is your name? Fred
Fred  is a great name!
===========
DATE: 2013-05-17
TIME: 16:08:14
==========++++++++++++++++++++++Success!
What is your name? Frank
Frank  is a great name!
===========
DATE: 2013-05-17
TIME: 16:08:24
==========++++++++++++++++++++++Success!
What is your name? Jerry
I don't like that name.
```

Participant 3

```
===========
DATE: 2013-05-20
TIME: 19:11:36
===========
  File "/Users/Marc/Desktop/p3/t4.py", line 7
    else
       ^
SyntaxError: invalid syntax
===========
DATE: 2013-05-20
```

```
TIME: 19:11:44
==========++++++++++++++++++++++Success!
Enter your name: Jerry
I don't like that name
==========
DATE: 2013-05-20
TIME: 19:11:47
==========++++++++++++++++++++++Success!
Enter your name: Fred
Fred is a name I like!
```

Task 5

Participant 1

```
1  ===========
2  DATE: 2013-05-17
3  TIME: 14:12:01
4  ===========++++++++++++++++++++++Success!
5  1
6  2
7  3
8  4
9  5
10 6
11 7
12 8
13 9
14 10
```

Participant 2

```
1  ===========
2  DATE: 2013-05-17
3  TIME: 16:11:28
4  ===========
5    File "/Users/Alex/Dropbox/study/p2/t5.py", line 3
6      i = 0
7      ^
8  IndentationError: expected an indented block
9  ===========
10 DATE: 2013-05-17
11 TIME: 16:11:55
12 ===========++++++++++++++++++++++Success!
13 0
```

```
1
2
3
4
5
6
7
8
9
10
```

Participant 3

```
============
DATE: 2013-05-20
TIME: 19:13:50
============++++++++++++++++++++++Success!
0
1
2
3
4
5
6
7
8
9
10
```

Task 6

Participant 1

```
1  ===========
2  DATE: 2013-05-17
3  TIME: 14:14:54
4  ===========++++++++++++++++++++++Success!
5  10
6  9
7  8
8  7
9  6
10 5
11 4
12 3
13 2
14 1
```

Participant 2

```
1  ===========
2  DATE: 2013-05-17
3  TIME: 16:15:34
4  ===========++++++++++++++++++++++Success!
5  10
6  9
7  8
8  7
9  6
10 5
11 4
12 3
13 2
```

```
1
0
```

Participant 3

```
===========
DATE: 2013-05-20
TIME: 19:15:10
===========++++++++++++++++++++++Success!
10
9
8
7
6
5
4
3
2
1
```

Task 7

Participant 1

```
1  ===========
2  DATE: 2013-05-17
3  TIME: 14:21:33
4  ===========
5    File "/Users/Alex/dropbox/study/p1/t7.py", line 11
6      def main():
7      ^
8  IndentationError: expected an indented block
9  ===========
10 DATE: 2013-05-17
11 TIME: 14:21:39
12 ===========
13 Enter a number5
14
15 Try a lower number
16 Try a lower number
17 Try a lower number
18 Try a lower number
19 Try a lower number
20 Try a lower number
21 Try a lower number
22 Try a lower number
23 Try a lower number
24 Try a lower number
25 Try a lower number
26 Try a lower number
27 Try a lower number
28 Try a lower number
29 ^CTry a lower number
30
```

```
Traceback (most recent call last):
  File "/Users/Alex/dropbox/study/p1/t7.py", line 18, in <module>
    main()
  File "/Users/Alex/dropbox/study/p1/t7.py", line 16, in main
    checkGuess(randomNum,guessNum)
  File "/Users/Alex/dropbox/study/p1/t7.py", line 6, in checkGuess
    print("")
KeyboardInterrupt
===========
DATE: 2013-05-17
TIME: 14:23:45
===========
  File "/Users/Alex/dropbox/study/p1/t7.py", line 18, in <module>
    main()
  File "/Users/Alex/dropbox/study/p1/t7.py", line 14, in main
    while randomNum != guessNum :
UnboundLocalError: local variable 'guessNum' referenced before assignment
===========
DATE: 2013-05-17
TIME: 14:24:29
===========
Enter a number between 1 and 10
Your number:5

Enter a number between 1 and 10
Your number:3

Enter a number between 1 and 10
Your number:2
You got it!
Enter a number between 1 and 10
Your number:^CTraceback (most recent call last):
  File "/Users/Alex/dropbox/study/p1/t7.py", line 22, in <module>
```

```
    main()
  File "/Users/Alex/dropbox/study/p1/t7.py", line 17, in main
    guessNum = int(input("Your number:"))
KeyboardInterrupt
===========
DATE: 2013-05-17
TIME: 14:27:22
===========
  File "/Users/Alex/dropbox/study/p1/t7.py", line 19
    if randomNum == checkGuess
                               ^
SyntaxError: invalid syntax
===========
DATE: 2013-05-17
TIME: 14:31:54
==========++++++++++++++++++++++Success!
Enter a number between 1 and 10
Your number:5

Enter a number between 1 and 10
Your number:2

Enter a number between 1 and 10
Your number:1
You got it!
```

Participant 2

```
===========
DATE: 2013-05-17
TIME: 16:28:19
===========
  File "/Users/Alex/Dropbox/study/p2/t7.py", line 5
    elif guessNo > randomNumber:
```

```
      ^
SyntaxError: invalid syntax
===========
DATE: 2013-05-17
TIME: 16:28:33
===========
Enter value between 1 and 10 5
Traceback (most recent call last):
  File "/Users/Alex/Dropbox/study/p2/t7.py", line 23, in <module>
    main()
  File "/Users/Alex/Dropbox/study/p2/t7.py", line 16, in main
    checkGuess(guessNo, randomNumber)
  File "/Users/Alex/Dropbox/study/p2/t7.py", line 5, in checkGuess
    if guessNo > randomNumber:
TypeError: unorderable types: str() > int()
===========
DATE: 2013-05-17
TIME: 16:29:52
==========++++++++++++++++++++++Success!
Enter value between 1 and 10
Enter value between 1 and 10 4
Too low
Enter value between 1 and 10 4
Too low
Enter value between 1 and 10 5
Too low
Enter value between 1 and 10 36
too high!
Enter value between 1 and 10 7
Too low
Enter value between 1 and 10 8
Too low
Enter value between 1 and 10 9
```

```
Wow you got it! It was  9
```

Participant 3

```
===========
DATE: 2013-05-20
TIME: 19:19:27
===========
  File "/Users/Marc/Desktop/p3/t7.py", line 6
    elif guessNo > randNo
                         ^
SyntaxError: invalid syntax
===========
DATE: 2013-05-20
TIME: 19:19:55
===========
  File "/Users/Marc/Desktop/p3/t7.py", line 16
    if guessNo = randomNo:
               ^
SyntaxError: invalid syntax
===========
DATE: 2013-05-20
TIME: 19:20:24
==========++++++++++++++++++++++Success!
Enter a guess: 1
Too low
Enter a guess: 5
Too low
Enter a guess: 8
Too high
Enter a guess: 7
Too high
Enter a guess: 6
Wow, you got it. It was  6
```

Task 8

Participant 1

```
==========
DATE: 2013-05-17
TIME: 14:36:38
==========
  File "/Users/Alex/dropbox/study/p1/t8.py", line 23, in <module>
    main()
  File "/Users/Alex/dropbox/study/p1/t8.py", line 21, in main
    if checkGuess(randomNum,guessNum) == true :
  File "/Users/Alex/dropbox/study/p1/t8.py", line 6, in checkGuess
    return true
NameError: global name 'true' is not defined
==========
DATE: 2013-05-17
TIME: 14:37:19
==========++++++++++++++++++++++Success!
Enter a number between 1 and 10
Your number:5

Enter a number between 1 and 10
Your number:2
You got it!
```

Participant 2

```
==========
DATE: 2013-05-17
TIME: 16:36:01
==========
  File "/Users/Alex/Dropbox/study/p2/t8.py", line 19
    if checkGuess(guessNo, randomNumber) = True
```

```
                                                ^
SyntaxError: invalid syntax
===========
DATE: 2013-05-17
TIME: 16:36:13
===========
  File "/Users/Alex/Dropbox/study/p2/t8.py", line 19
    if checkGuess(guessNo, randomNumber) == True
                                                 ^
SyntaxError: invalid syntax
===========
DATE: 2013-05-17
TIME: 16:36:33
===========++++++++++++++++++++++Success!
Enter value between 1 and 10: 4
Too low
Enter value between 1 and 10: 4
Too low
Enter value between 1 and 10: 5
Too low
Enter value between 1 and 10: 6
Too low
Enter value between 1 and 10: 7
Wow you got it! It was  7
```

Participant 3

```
===========
DATE: 2013-05-20
TIME: 19:22:26
===========
Enter a guess: 1
Too low
Enter a guess: 3
```

```
8  Too high
9  Enter a guess: 2
10 Wow, you got it. It was  2
```

Task 9

Participant 1

```
1  ===========
2  DATE: 2013-05-17
3  TIME: 14:41:20
4  ===========
5    File "/Users/Alex/dropbox/study/p1/t9.py", line 10, in <module>
6      main()
7    File "/Users/Alex/dropbox/study/p1/t9.py", line 4, in main
8      Rainbow[Red,Orange,Yellow,Green,Blue,Indigo,Violet]
9  NameError: global name 'Rainbow' is not defined
10 ===========
11 DATE: 2013-05-17
12 TIME: 14:41:50
13 ===========
14   File "/Users/Alex/dropbox/study/p1/t9.py", line 10, in <module>
15     main()
16   File "/Users/Alex/dropbox/study/p1/t9.py", line 4, in main
17     Rainbow = [Red,Orange,Yellow,Green,Blue,Indigo,Violet]
18 NameError: global name 'Red' is not defined
19 ===========
20 DATE: 2013-05-17
21 TIME: 14:42:40
22 ===========
23   File "/Users/Alex/dropbox/study/p1/t9.py", line 10, in <module>
24     main()
25   File "/Users/Alex/dropbox/study/p1/t9.py", line 5, in main
26     i = int(o)
27 NameError: global name 'o' is not defined
28 ===========
29 DATE: 2013-05-17
30 TIME: 14:43:28
```

```
31 ===========
32   File "/Users/Alex/dropbox/study/p1/t9.py", line 10, in <module>
33     main()
34   File "/Users/Alex/dropbox/study/p1/t9.py", line 7, in main
35     print(Rainbow[i])
36 TypeError: list indices must be integers, not str
37 ===========
38 DATE: 2013-05-17
39 TIME: 14:43:40
40 ===========
41   File "/Users/Alex/dropbox/study/p1/t9.py", line 10, in <module>
42     main()
43   File "/Users/Alex/dropbox/study/p1/t9.py", line 8, in main
44     i+=1
45 TypeError: Can't convert 'int' object to str implicitly
46 ===========
47 DATE: 2013-05-17
48 TIME: 14:45:00
49 ===========
50   File "/Users/Alex/dropbox/study/p1/t9.py", line 8
51     +=1
52      ^
53 SyntaxError: invalid syntax
54 ===========
55 DATE: 2013-05-17
56 TIME: 14:45:30
57 ===========
58 ['Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Indigo', 'Violet']  is in a
      rainbow
59 ['Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Indigo', 'Violet']  is in a
      rainbow
60 ['Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Indigo', 'Violet']  is in a
      rainbow
```

```
['Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Indigo', 'Violet']  is in a
   rainbow
['Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Indigo', 'Violet']  is in a
   rainbow
['Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Indigo', 'Violet']  is in a
   rainbow
['Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Indigo', 'Violet']  is in a
   rainbow
===========
DATE: 2013-05-17
TIME: 14:45:47
===========
  File "/Users/Alex/dropbox/study/p1/t9.py", line 9, in <module>
    main()
  File "/Users/Alex/dropbox/study/p1/t9.py", line 7, in main
    print(Rainbow[i]," is in a rainbow")
TypeError: list indices must be integers, not str
===========
DATE: 2013-05-17
TIME: 14:46:14
===========++++++++++++++++++++++Success!
Red  is in a rainbow
Orange  is in a rainbow
Yellow  is in a rainbow
Green  is in a rainbow
Blue  is in a rainbow
Indigo  is in a rainbow
Violet  is in a rainbow
```

Participant 2

```
===========
DATE: 2013-05-17
TIME: 16:45:21
```

```
==========
  File "/Users/Alex/Dropbox/study/p2/t9.py", line 4
    for i in Rainbow
                   ^
SyntaxError: invalid syntax
==========
DATE: 2013-05-17
TIME: 16:45:30
==========
  File "/Users/Alex/Dropbox/study/p2/t9.py", line 7, in <module>
    main()
  File "/Users/Alex/Dropbox/study/p2/t9.py", line 5, in main
    print(Rainbow[i])
TypeError: list indices must be integers, not str
==========
DATE: 2013-05-17
TIME: 16:46:19
==========++++++++++++++++++++++Success!
Red
Orange
Yellow
Green
Blue
Indigo
Violet
```

Participant 3

```
==========
DATE: 2013-05-20
TIME: 19:25:30
==========
  File "/Users/Marc/Desktop/p3/t9.py", line 4
    for (i) in rainbow
```

```
                      ^
SyntaxError: invalid syntax
===========
DATE: 2013-05-20
TIME: 19:25:43
===========
  File "/Users/Marc/Desktop/p3/t9.py", line 4, in <module>
    for (i) in rainbow:
NameError: name 'rainbow' is not defined
===========
DATE: 2013-05-20
TIME: 19:28:36
==========++++++++++++++++++++++Success!
Red
Orange
Yellow
Green
Blue
Indigo
Violet
```

Task 10

Participant 1

```
1  ===========
2  DATE: 2013-05-17
3  TIME: 14:51:22
4  ===========
5    File "/Users/Alex/dropbox/study/p1/t10.py", line 15, in <module>
6      main()
7    File "/Users/Alex/dropbox/study/p1/t10.py", line 12, in main
8      sumArray(numbers)
9    File "/Users/Alex/dropbox/study/p1/t10.py", line 4, in sumArray
10     i,runningTotal = int(0)
11 TypeError: 'int' object is not iterable
12 ===========
13 DATE: 2013-05-17
14 TIME: 14:52:55
15 ===========
16   File "/Users/Alex/dropbox/study/p1/t10.py", line 15, in <module>
17     main()
18   File "/Users/Alex/dropbox/study/p1/t10.py", line 12, in main
19     sumArray(numbers)
20   File "/Users/Alex/dropbox/study/p1/t10.py", line 4, in sumArray
21     i,runningTotal = 0
22 TypeError: 'int' object is not iterable
23 ===========
24 DATE: 2013-05-17
25 TIME: 14:54:20
26 ===========
27   File "/Users/Alex/dropbox/study/p1/t10.py", line 15, in <module>
28     main()
29   File "/Users/Alex/dropbox/study/p1/t10.py", line 13, in main
30     print(runningTotal)
```

```
NameError: global name 'runningTotal' is not defined
===========
DATE: 2013-05-17
TIME: 14:55:31
===========++++++++++++++++++++++Success!
30
```

Participant 2

```
===========
DATE: 2013-05-17
TIME: 16:51:32
===========++++++++++++++++++++++Success!
The total is  30
```

Participant 3

```
===========
DATE: 2013-05-20
TIME: 19:31:30
===========++++++++++++++++++++++Success!
30
```

C Raw Data

This Appendix lists the raw data calculated for the results of this study. They are attached in the following order:

1. Pre-Evaluation Survey Data
2. Post-Evaluation Survey Data
3. Observation Data