Unit 2: Computational thinking

Introductory task

Learning objectives

- Introduction to the thinking skills required to solve computational problems

Introduction

Computational thinking is what allows us to go from the quick and simple little programs we've been writing, to solving more meaningful problems.

Discussion & learning items

When discussing Computational Thinking, there are four thinking skills commonly associated with this.

- Decomposition - Can I divide this into sub-problems?
- Pattern recognition - Can I find repeating patterns?
- Abstraction - Can generalise this to make an overall rule?
- Algorithm design - Can I document the programming steps for any of this?

To demonstrate Computational thinking in action, study the walk through in these slides:

- Slides: https://pbaumgarten.com/igcse-compsci/igcse-2-computational-thinking-slides.pdf
- Video with narration of the process: https://youtu.be/2bvt6PCBVPo

Finally, some other general advice I would give to new programmers is:

- Just start - A blank screen can be overwhelming
- Don’t start at the start - Programs are not novels, you read and write a program by jumping around
- Start with something you know - Build the user interface and work back from there
- Don’t be afraid to Google - Prioritise results from forums such as stackoverflow
- Test & print a lot

Exercises

Once you've digested the slides or video, have a go at a couple of problems of your own:

Problem 1: Create an age calculator. Prompt the user for (1) their birthdate and (2) the current date, calculate their age, checking whether they have had their birthday yet or not.

Problem 2: Create a money change calculator. Given a set of possible coins available to the shop attendant, and an amount of change they must provide the customer, calculate how many of each coin they should give the customer. For example if a country has 1cent, 5cent, 10cent, & 50cent coins and you need to provide 67cents of change, the clerk would give 1x50cent, 1x10cent, 1x5cent, and 2x1cent coins.

Further practice & resources

https://pbaumgarten.com/problems/

2. Introducing systems & algorithms

Learning objectives

- Introduction to the idea of top-down design and structure diagrams
- Practice identifying subsystems and document it on a structure diagram
- Appreciation of the need to document algorithms

Introduction

Pre-reading from textbook

- 9.1 Introduction
- 9.2 Algorithms

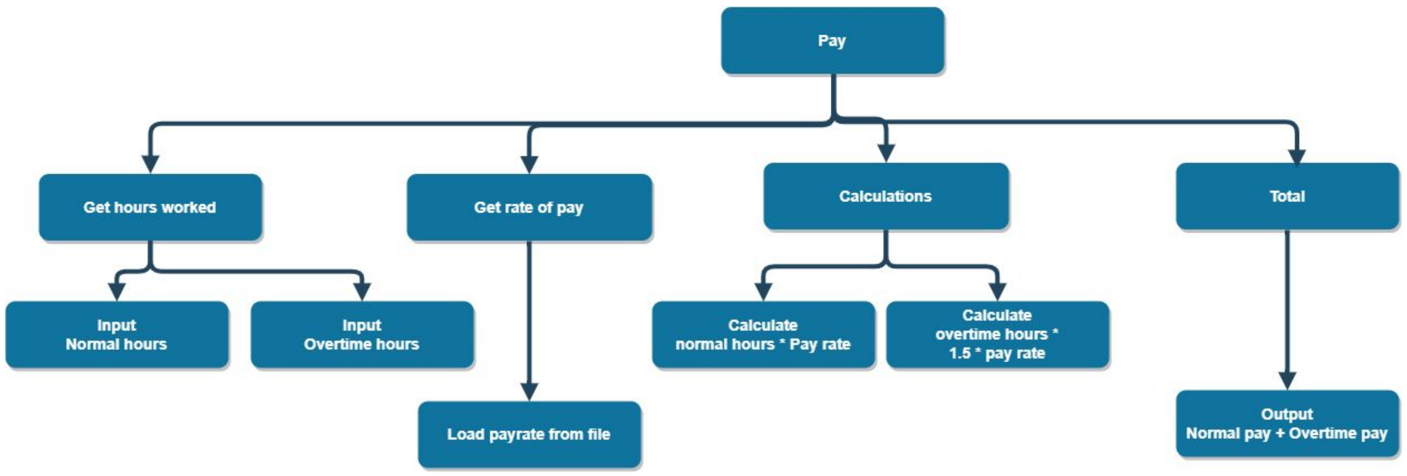
Discussion & learning items

Top down design

Top-down design is another way of thinking about the process of decomposition. To quote from the textbook, "each computer system can be divided up into a set of subsystems. Each sub-system can be further divided into subsystems and so on until each sub-system just performs a single action" (p115).

This idea of dividing a system into its subsystems is known as top-down design, where at the apex you have your overall project or system, and then you create a hierarchical tree of subsystems from that.

This hierarchy can be drawn into a chart, which is known as a **structure diagram**. An example is provided below. There is also an example for an alarm app in the textbook on page 116.



The main question that might arise is "how do I know when I need to divide a system into new sub-systems"? For the purposes of this course, it is fairly safe to think of each box as a function. If it makes sense to write new functions, you are creating new sub-systems.

Documenting algorithms

Algorithm: "a sequence of instructions, typically to solve a class of problems or perform a computation" (wikipedia). So, it is just a technical term for a set of steps or instructions. It is analogous to a recipe for cooking.

So, once we know what an algorithm is, the next question is how to write one. Writing computer code is an act of writing an algorithm. You are creating a series of steps for that programming language's compiler or interpreter to follow. It is also very useful to be able to write algorithms for other computer scientists to understand, even if they aren't a specialist in the same programming language you use. A couple of commonly used tools for this are flowcharts and pseudocode.

You've probably encountered flowcharts before. They form part of this course, so we will get more familiar with them over the coming lessons. For now, look at the simple example on page 115.

Pseudo-code is a term used for a "structured English" that looks like and takes on the shape of a programming language, but where the instructions are written in English. Again, check the simple example on page 115 but we will discuss it more in lessons to come.

The overall idea here is that through a combination of an overall structure diagram, and then creating either pseudo-code or flowcharts for each sub-system identified, we can fully document the design of any project.

Exercises

1. Complete Activity 9.2 (don't bother with 9.3, it's a bit lame)
2. Draw a structure diagram for the sort algorithm we discussed in the computational thinking lesson. (example solution)
3. Draw a structure diagram for the age calculator or change calculator from the previous lesson.
4. Working with a partner, pick one of the problems from my programming problems page @ https://pbaumgarten.com/problems (such as Hang-person or Tic-tac-toe) and identify the subsystems that would be involved. Draw a structure diagram for that problem.
5. For one of the above, experiment with what you think the flowchart to document each identified sub-system might look like.

3. Trace tables

Learning objectives

- A trace table is a manual mechanism of debugging an algorithm

Introduction

Read 9.5 Using trace tables and
Read 9.6 Identifying and correcting errors

Discussion & learning items

Trace tables are a manual tool for testing and correcting an algorithm.

They are a table where you manually walk through each line of your algorithm, calculating the value of each variable as you go. Every variable gets its own column, and you add a new row for every new step of program execution.

Exercises

Activity 9.14 through 9.17 (textbook 127-129)

4, 5. Tracing pseudocode

Learning objectives

- Pseudocode is a manner of documenting a computer algorithm in a way that is programming-language agnostic, so it can be understood by any programmer no matter their language of speculation.

Introduction

Read 9.7
Read all of chapter 10

Discussion & learning items

What is pseudo code?

"Pseudo" is defined as "not actually but having the appearance of; pretended; false or spurious; sham; almost, approaching, or trying to be." (reference).

So pseudo-code is pretend-code. It is not a genuine, real programming language. The intent behind the idea is that it is a generic, language agnostic tool that can be used to communicate algorithms between programmers regardless of what language they may be specialists in. That is, something that can be read and understood by all programmers because it is so clear and simple.

The following is an example of pseudo-code you will see in this course. You should be able to read it, understand what it is conveying and convert it into your own programming language without difficulty.

```
PRINT "I can count! What number should I count up to?"
INPUT Target
N ← 1
WHILE N < Target
   PRINT N
   N ← N + 1
PRINT "Told you I could do it!"
```

A note about pseudo code: What it is supposed to be versus what it is in the iGCSE course... While the idea of pseudo-code is supposed to be a language-agnostic, generic undefined tool for documenting algorithms, that is not compatible with the notion of exam marking schemes. As a result, pseudo code for our course is actually quite prescriptively defined. While you don't have to be syntactically perfect, you are expected to write pseudocode that is very clear and to the same level of detail as defined within the syllabus. In other words, while not a real programming language, you do need to learn it and practice it for it will appear in your exams.

The official syntax of pseudo code for the iGCSE course:

- Chapter 10, page 134-139 of the textbook.

Exercises

Practice your knowledge of trace tables and pseudo code.

Problem 1.

Create a trace table for the following algorithm to determine its purpose.

```
OUTPUT "Enter the first integer: "
INPUT x
OUTPUT "Enter the second integer: "
INPUT y
z ← 0
WHILE x > 0
    IF x mod 2 = 1 THEN
        z ← z + y
    ENDIF
    x ← x div 2
    y ← y * 2
ENDWHILE
OUTPUT "Answer =", z
```

Problem 2.

A doctor records a patient's temperature once an hour for six hours. Any time the temperature is > 37C, an incidence of fever is recorded. The average of all temperatures taken is calculated at the end.

```
temp ← 0
fever ← 0
total ← 0
hour ← 1
WHILE hour < 7 THEN
    OUTPUT "Enter temperature: "
    INPUT temp
    IF temp > 37 THEN
        fever ← fever + 1
    END IF
    total = total + temp
    hour = hour + 1
ENDWHILE
average ← ROUND(total/hour,1) # round to 1 decimal place
OUTPUT "Average temperature:", average
OUTPUT "Incidents of fever:", fever
```

Problem 3.

Complete a trace table to determine the final state of the variables in this algorithm.

```
sum ← 0
N ← 10
WHILE N < 40
   sum ← sum + N
   OUTPUT N, SUM
   N ← N + 5
END WHILE
```

Problem 4.

Use a trace table to determine what is printed by the following when N = 5.

```
IF (N = 1) OR (N = 2) THEN
   H ← 1
ELSE
   F ← 1
   G ← 1
   FOR J ← 1 TO N-1 DO
      H ← F + G
      F ← G
      G ← H
      OUTPUT H
   END FOR
END IF
OUTPUT F, G
```

Problem 5.

Use a trace table to determine what is printed by the following.

```
Sum ← 0
N ← 20
WHILE N < 30
    Sum ← Sum + N
    OUTPUT N, SUM
    N ← N + 3
END WHILE
```

Problem 6.

Use a trace table to determine what is printed by the following.

```
Count ⟵ 1
X ⟵ 2
WHILE Count < 25
    X ⟵ X + 2
    OUTPUT Count, X
    Count ⟵ Count + 5
END WHILE
```

Problem 7.

Complete a trace table for the following given that the input will be 6.

```
INPUT X
FOR M = 1 TO X
    Y ⟵ X - M
    Z ⟵ 5 * Y - M
END FOR
OUTPUT Z
```

Credits:

- Problem 1 & 2: https://www.tes.com/teaching-resource/trace-table-activities-for-gcse-computer-science-11782527
- Problem 3 onwards: https://retrosnob.files.wordpress.com/2013/11/trace-tables-chapter.pdf

6. Writing pseudocode

Learning objectives

- Being able to write good pseudocode is just as important as being able to read it.

Introduction

Time to practice writing some of your own pseudo code.

Exercises

1. User inputs the volume of a sphere, the program outputs the radius.
2. User inputs two numbers, A and B, the program will output all the powers of A, e.g. A*1, A*A, A*A*A, up to the value of B. B can't be less than 2.
3. User inputs two positive numbers, the program counts up from the smaller of these numbers to the larger, outputting the numbers as it counts up.
4. User inputs a number between 1 and 99. Program counts from 1 to the input number in word form (eg: one, two, three, four...).

Further practice & resources

70 pseudocode practice questions:

https://pbaumgarten.com/igcse-compsci/distribute/pseudocode-70-questions.pdf

(I have the solutions for these for you to self-check)

7, 8. Tracing & writing flow charts

Learning objectives

- Flowcharts is an alternative to pseudocode for the documenting of algorithms.

Introduction

Pre-read chapter 10, page 142 of the textbook.

Discussion & learning items

Just like pseudo-code, flowcharts are intended to be a language-agnostic way of communicating algorithms from one programmer to another. Also just like pseudo-code, the course syllabus has mandated a set of symbols for you to learn.

Syntax of flowcharts for the iGCSE course:

- Chapter 10, page 142 of the textbook.

It is important to pay attention to your use of the symbols and be syntactically correct.

Exercises

1. Draw a flowchart that inputs 20 numbers and outputs the range between the highest and lowest number (without using min() or max()).
2. Draw a flowchart that inputs 10 numbers and calculates the average of those that were between 0 and 100.
3. Draw a flowchart that inputs the names of 10 people, and prints the name of the person with the longest name.

Further practice & resources

Any of the 70 pseudocode questions previously provided could just as easily be converted into a flowchart question. Pick one or two of those that you haven't done yet.

8. Testing algorithms

Learning objectives

- Appreciation for the need to test that our algorithms do not fail when presented with unexpected data

Introduction

Pre-reading from textbook: 9.3 Test data

Discussion & learning items

A maxim of computer science is **do not trust the user!** If they can misinterpret your instructions (or intentionally try to work around them), they will. The corollary of that is to **always parse your inputs!** We do this through verification and validation.

As our programs become more complicated, having a proper testing regime becomes increasingly important. It is no longer good enough to run our program with a "typical" value, get the expected result and consider it "working".

A well designed testing regime will consider four different types of input data:

- Normal data: This is the data you have designed your program to receive as input..
- Erroneous & abnormal data: Data your program should reject if received as input.
- Extreme data: The extreme values of what is valid to receive as an input.
- Boundary data: Data on the edge of what your program may receive as inputs. This tends to overlap with the extreme data. You include data just inside the boundary (test it is accepted) and data just outside the boundary (test it is rejected).

Example: An alarm app that requires you to enter a time for the alarm in 24 hour format, hh:mm.

Data type	**Examples**
Normal data	06:30, 07:00, 16:00
Erroneous data	09:00am, Seven, -17:00, 4 o'clock, 25:00, 8:75, 5pm
Extreme data	00:00, 23:59
Boundary data	00:00, 23:59, 24:00, 05:60

(Of course, you could modify the app's requirements so that it can correctly interpret some of the "erroneous data" so it can become "normal data")

The question then becomes: how do you want your app to respond to the various types of data? And does it respond in the manner you intend? This is what is required of a proper testing regime.

It may also be necessary to state some assumptions when devising your test data if the problem scenario is not sufficiently clear.

Practice

A great example of the need for this kind of testing is from our “Making decisions” problem set question 2. About 40% of students submitted something like the following. If this code is properly tested, an error would be discovered.

Create a set of testing data (normal, erroneous, boundary) to seek to identify how this program does not behave in the manner you might expect.

```python
side_a = int(input("Input side A of your triangle: "))
side_b = int(input("Input side B of your triangle: "))
side_c = int(input("Input side C of your triangle: "))

if side_a^2 + side_b^2 == side_c^2:
  print("This is a pythagorean triple.")
else:
  print("This is not a pythagorean triple :(")
```

Exercises

- Activity 9.5, 9.6, 9.7, 9.8 (Textbook pages 119, 120).
- Create a table and generate the four types of test data for the following problems:
    a. Currency input for a currency conversion app that will convert HKD to EUR.
    b. Date input for a date reminder app that will accept dates (in the style of dd/mm/yyyy, or an alternative you may prefer).
    c. Hong kong mobile phone number input for an authentication app that will send SMS messages (hint: check wikipedia for the rules of what constitutes valid mobile phone numbers in HK)
    d. Email address input
    e. Postal address input

9. Validation basics

Learning objectives

- Consider the importance of validating input from the user before accepting it into your program
- Implement input validation checks for some given problems

Introduction

- One way we can minimise problems when presented with unexpected data, is to **validate** the data input before attempting to use it in our algorithm.

Discussion & learning items

Pre-read from textbook: 9.4 Validation and verification

Validation checks are the process of programmatically checking the input given to your program to ensure it meets basic criteria. The idea is to check what the user has input to detect possible mistakes before accepting it within your program.

Some of the typical checks involved are:

- Range checks - Is the input value within the permitted minimum and maximum range?
- Length checks - Is the input value within the permitted size length? eg: perhaps you only allow up to 30 characters for an email address. You will have encountered these kinds of limits on apps and websites many times, why do they exist? Typically it's because these systems save their data into a database, and databases require you to specify the size of each field so it can allocate the appropriate amount of disk space.
- Type checks - Is the input value of the correct type? ie: if you ask for an integer, but are given a string or a float.
- Character checks - Does the input value only contain permitted characters? Sometimes special punctuation symbols can cause problems.
- Format checks - Is the style of the input value according to expected rules? eg: dd/mm/yyyy format for a date.
- Presence checks - Has the input been provided if it is required? eg: not left blank.
- Algorithmic checks - We will consider this next lesson.

In the previous lesson we used a range of possible app ideas and the inputs they would require. The first of these was an alarm code. To create the validation code for a valid 24 hour time, I came up with the following…

```
def check_valid_time(time):
    # Presence check
    # Length check - must be 5 characters
    if len(alarm) != 5:
        return False
    # Format check - look for colon character
    if alarm[2:3] != ":":
        return False
    # Type check - Must be numbers
    # Format check - numbers in correct positions
    if not alarm[0:2].isnumeric():
        return False
    if not alarm[3:5].isnumeric():
        return False
    # Range checks
    hours = int(alarm[0:2])
    minutes = int(alarm[3:5])
    if hours < 0 or hours > 23:
        return False
    if minutes < 0 or minutes > 59:
        return False
    # All checks passed
    return True

# Testing code
alarm = input("Enter alarm time as hh:mm >> ")
ok = check_valid_time(alarm)
print(ok)
```

Problems

In the previous lesson, you created testing data for a range of common input scenarios. They were:

- Currency input for a currency conversion app that will convert HKD to EUR.
- Date input for a date reminder app that will accept dates (in the style of dd/mm/yyyy, or an alternative you may prefer).
- Hong kong mobile phone number input for an authentication app that will send SMS messages (hint: check wikipedia for the rules of what constitutes valid mobile phone numbers in HK)
- Email address input
- Postal address input

Select TWO of the above scenarios and create Python functions that would seek to validate their input.

10, 11. Validation using algorithms

Learning objectives

- Investigate the role of check digits to validate input
- Implement a common check digit algorithm

Introduction

Once you've validated the user input with range checks, format checks, length checks and so forth, the user may have still entered incorrect data. Humans are rather error-prone when recording numbers. This is because there aren't set rules for noticing when you have incorrectly written a number like a spelling error would make obvious for written words.

For some special types of data, such as bank account numbers, it might be worth the extra hassle of one more additional check.

This additional check involves creating an algorithmic rule that allows the data to be used to check itself. This occurs by adding what is known as a **check digit**. This check digit can be automatically calculated from the rest of the number. If the calculation works, then there is a much greater likelihood that the number has been correctly entered (though mistakes can still occur).

Discussion & learning items

Check digits are used whenever it is considered important to minimise data entry errors of numbers.

Three common numbers you may be familiar with that contain check digit algorithms are:

- ISBN - The last digit of the ISBN is a check digit. The algorithm is explained on page 122 of the textbook.
- Most common credit cards use the LUHN algorithm to generate a check digit. The algorithm is documented at....
    - https://pbaumgarten.com/problems/problem-luhn.pdf
- The Hong Kong ID number is another check digit algorithm. The number in the bracket is generated by the algorithm. The algorithm is documented at...
    - https://access-excel.tips/hkid-check-digit/

Exercises

Working in pairs or individually, select one of the problems: ISBN, LUHN or HKID.

1. Create a table to devise test data (normal, erroneous, extreme, boundary)
2. Decide on the input checks you will use (range, length, type, character, format, presence)
3. Use computational thinking to decide how you will calculate the check digit (decomposition, abstraction, pattern recognition, algorithm design)
4. Code it in Python
5. Use your test data - do your checks behave as they should?
6. Submit your written work (tests and checks) and Python programming.

12. Verification

Learning objectives

- Verification is an additional measure that can be used to ensure the accuracy of data before using it for processing.
- Verification involves having the user check the data

Introduction

Validation checks was about the program conducting its own checks to identify erroneous data entry. Verification checks are about involving the user to manually check the input themselves and confirm it as correct.

Be aware this distinction between validation and verification can sometimes get a bit confused in industry and when reading resources online. The above difference is how the iGCSE course defines them so it is what you need to know. As wikipedia puts it so well, "In practice... the definitions of verification and validation can be inconsistent. Sometimes they are even used interchangeably". Be aware of this issue when searching online.

Discussion & learning items

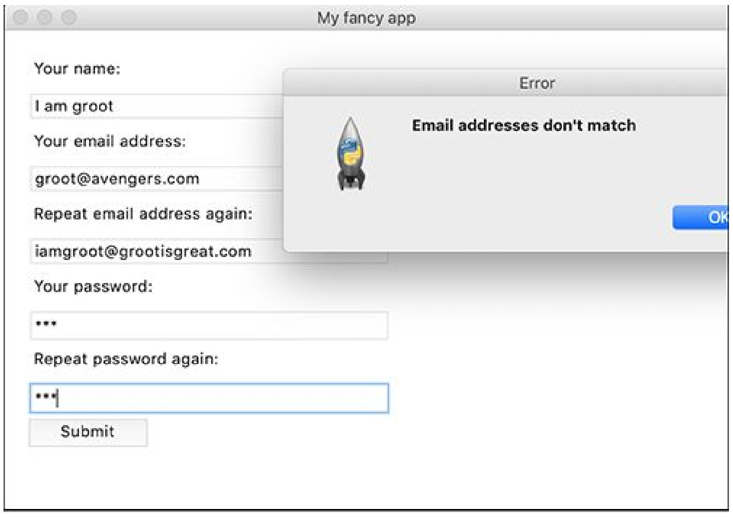
Verification is about ensuring the user has typed what they intended to type. There are a few ways a programmer could do this. Some common strategies include:

- Double entry - Have the user enter the information twice. This is used a lot with passwords and email addresses.
- Screen/visual check - Show the user the information entered and ask them to click to confirm it is correct. This is used a lot with online shopping prior to finalising a purchase.
- Check digit (notice that this can be used for validation and verification)

Exercises

As an exercise in using verification, I have provided the code to create a Windows application that will ask for email and password fields to be entered twice. This makes an ideal excuse to expose you to what is involved in creating a Windows style GUI with Python.

The provided code will get the windows GUI system up and running, but you will need to finish the **`clicked()`** function to perform the verification step.



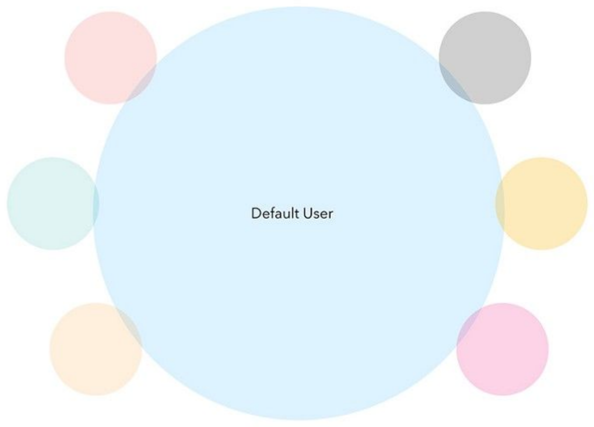
https://pbaumgarten.com/igcse-compsci/distribute/unit-2-tkinter-verification-demo-app.pdf

Further practice & resources

If you are interested in learning more about GUIs with Tkinter, check:

- The last chapter of my Python book
- https://pbaumgarten.com/python/tkinter.md
- I have a video series of 5 lessons about Tkinter on my Youtube channel

13. Edge cases are people too



"The humans who sit on the margins of our products are the same humans who sit on the margins of society."

As companies push for scale and growth at breakneck pace they are weaponizing technology against groups that fall outside of their defined happy path.

The "happy path design" is not human-centered, it is business-centered. It's good for businesses because it allows them to move fast.

Designing for speed has trained us to ignore edge cases, and the overwhelming prevalence of homogenous teams made up of the least vulnerable among us (read: dudes) has conditioned us to center their life experience in our design process.

Today we are building massive platforms, with massive reach and impact, yet they are massively fragile. If we are honest with ourselves, these platforms represent a failure of design. Their success hinges on an intentional disregard for human complexity, and society pays the price for it.

Facebook claims to have two billion users…1% of two billion is twenty million. When you're moving fast and breaking things 1% is well within the acceptable breaking point for rolling out new work. Yet it contains twenty million people. They have names. They have faces. Technology companies call these people edge cases, because they live at the margins. They are, by definition, the marginalized.

While it may never have been explicitly stated, the inherent bias in our product development approach has meant that our underlying first principle has been to ignore human complexity.

Reference: https://modus.medium.com/who-pays-the-price-of-happy-path-design-36047f00c044

Discussion questions

There are many subtle complexities that can be easily washed over in the rush to bring a product to market:

- Names
- Gender & sex
- Language
- Assumed wealth

1. What are some ways in which software could marginalise people by making assumptions about the above categories?

2. What other categories can you identify in which software marginalises people? Examples?

3. What steps could you take as a programmer to help ensure your products don't further marginalise people?

Mini project: Computational thinking practice

A reminder of the summary of computational thinking

Decomposition	Pattern recognition	Abstraction	Algorithm design
Divide the problem into smaller, manageable pieces	**Identify repeating patterns**	**Focus on what matters.**	**Develop the step by step procedure that solves the problem**
Complete a brainstorm, identify parts to your problem. Do a manual walk with a small scale, human solvable version of the problem. ("trace table") What values will your variables have, what steps do you need to perform?	This is where you will discover loops and abstractions. Consider the inner part of any loop as a new problem to decompose, abstract and algorithmically design.	Strip away unnecessary detail that distracts from the actual problem.. Identify your variables, functions and classes.	Pseudo code or flowcharts can be helpful. Start (a blank screen can be overwhelming) Don't start at the start (It's not a novel) Start with something you know (such as the UI) Test & print a lot. Test your proposed solution works on larger, real-world versions of the problem.

1. Draw a brain storm of the problem identifying sub-parts to the problem. For each part, identify:
    a. Likely variables
    b. Likely constants
    c. Likely programming instructions
2. Create a set of example data. Solve the problem manually. What are the different steps you took? What were the calculations involved? What were the values created during the calculation (these will become your variables)
3. Document your procedures via pseudocode or flowchart.
4. Test your procedures works with a different set of example data.
5. Turn your procedures into code comments.
6. Now… Program your procedures
7. Test with expected and unexpected data. Remember to print a lot while testing.

19. Quiz

Some review questions can be found at:

- Chapter 9 review questions: Page 132-133.
- Chapter 10 review questions: Page 144-145.

The unit test will assess you on:

- Structure charts for top-down design of subsystems
- Creating & using test data
- Validation and verification of input
- Trace tables
- Pseudo code
- Flow charts