1 Jun 18

A possible idea I came up with was to program a basic calculator, complete with UI. It’s kind of silly, because I can do that in Excel already, not to mention having actual calculators on all my devices and even the actual device itself. Yet perhaps that’s what makes it precisely the best first goal to set. Another one would be to do a difference engine. Both are actually a bit more challenging than they seem because I don’t actually know how to produce an interface on demand in Python, nor do I know how to declare a function. Such tasks I can do in C++ or Java, oddly enough.

That may be an idiosyncrasy of the website I’m on. A proper course would probably have exactly those exercises in store. Ah well, it seems like there’s considerable overlap anyway, so I’ll continue with this fairly comprehensive guide for now. And there are still advanced tutorials waiting for me…

At least I learned to start from the contents-down in terms of learning, and also did a quick overview of the basic tutorials. It seems that the data type articles are worth diving into. Personally I thought dictionaries would be the most intriguing as a subset of lists, which are pretty powerful themselves.

Of course there are plenty of other programs that deal with matrices better, namely, MATLAB. We’ll see if I can get to that. But can I code a mod by the end of summer? It really depends. Conservatively I’ll say I need a couple of weeks to familiarise myself with Python. Less if I put in the effort unlike the weeks before.

2 Jun 18

//I realised one of my problems with Python: It runs code instantly. So it becomes very hard to keep track of the entire code. But I’m sure there’s a way to work with an entire function. The first one I might try is computing sine or *e*, which have infinite series.

Also finally dove into the programming sections of *CC:CS*. I literally just finished watching the section on data structures: arrays, matrices, linked lists and trees. Very useful because when I learned C++ I got stuck on linked lists.

//I’ve been thinking about my road to a Minecraft mod more and more. I have a few mod ideas: Slimycraft for content, a savegame one for utility, and the holy grail would be a standalone export of NEI/JEI recipes.

So right now my professional path would be Python 🡪 Java 🡪 R/MATLAB.

The gaming path would then be Python 🡪 Java 🡪 Forge 🡪 Github 🡪 Minecraft. Forge certainly is something else I’ll admit, and Github was less intuitive than I expected when I last tried it.

Anyway, I found the def method for Python. Remains to be seen if I can use it good enough. So far I haven’t actually seen an IO module for Python.

20 Jul 18, F

I did something really nuts. I got bitten by the headswap bug big time since the drawings of Sammie Oak. But I was too lazy to actually draw. So I went to some dressup games, downloaded them and cobbled them together. Well, at first I just took the images. Then I had the bright idea of building my own game. After all, it’d just be your regular game only with a beautiful boy as the model.

Little did I know that I’d be downloading Adobe Animate, working with Actionscript and doing vector art to upgrade a game to Actionscript 3.0. Honestly, the dream is to have totally customisable colours on pre-defined clothes, faces and facial features. And yes, the can-can boy is indeed beautiful, he’s so lucky that the serum worked so well on him. That will be complicated to explain in the portfolio, unless I briefly tidy it up.

What was really important was the reminder that I could use layers to separate the art. That was during my time of insanity in re-colouring five hairstyles in five more colours. But that’s simple compared to my next problem.

The way I was taught to link the buttons and outfits is kind of convoluted. The outfit frames lie within a unique handler symbol. Frames within the handler are exclusive, but frames between handler can stack. (In fact a certain dress’s stockings stack in front of the shoes, giving me another minor headache.) The issue arises with accessories that should be able to stack together. This means each of these individual items will have its own handler and own frames within the handler. OK, the handlers and frames have to be created.

More than that though, each accessory button has to have its own button mode activated. That will take a loop through an accessory array, not too hard. But the next part needs a specific mouse function that will call on that specific handler. The handler at least must change its name each time. Technically the function need not, but the index in the function array must change.

So three arrays, each for the handler, the accessory and the mouse function. It’s really hard to compile in Animate because too many things can break. I think I can do it in Python instead. Think of the three arrays as holding unique variables that have to be linked only with each other.

Also, I managed to nest arrays together to handle all the different combinations of hairstyles. That felt good, because I didn’t think it was possible.

edX: Intro to ComSci using Python

Algorithm

Program

Declarative statements

Imperative statements

Fixed program computer

Stored program computer

Primitives

Parameters

Abstraction

Turing completeness - Anything that can computable in one language is computable in another language.

**Languages**

Operations

Expression – Complex but legal combinations of primitives in a programming language

Value

Syntax – Arrangement of words. Used to assess if an expression is legal

Static semantics – When syntactically valid strings have meaning.

Semantics – Meaning associated with a syntactically valid string of symbols without errors. But errors can still occur due to various interpretations of a symbol.

Associated with the various kinds of errors that can arise.

Computational modes of thought

Program – a sequence of definitions and commands  
Definitions are evaluated – Assignment of names to values  
Commands are executed (by Python interpreter within a shell)

Objects – Contain data of different types  
Scalar – Cannot be subdivided i.e. atomic  
Non-scalar – Can be broken down

Int, Float, Bool, NoneType

Casting – Converting data objects between types

Syntax of an expression: <obj> <op> <obj>

Binding variables

Branching program

Test

Conditional

Loops – Iterative algorithms, Control flow  
For and while

**For** is used for known number of iterations. May be rewritten as while. Also uses a counter

**While** is used when the loop will run an unknown number of times. Uses a counter but initialisation is required.

For x in range(start, end, step):

To decrement for loops, set the start > end and a step of negative value.

More elegant than traversing a pre-defined array.

Python editor can be opened as a new window from the shell in IDLE.

I’ve been thinking about why I can’t just use Excel for all this arithmetic. The answer lies in the method of data storage. In Excel, the data has to be pre-defined, and entered into the cells. On the other hand, with loops and iterations we can generate such data on the fly. Also, Excel doesn’t handle logic all that well, plus we never have an overview of the code as a whole. Each cell contains its own code.

Plus the lecturer has indeed been recapitulating basic arithmetic. If that’s good enough for him, that’s good enough for me. But it is boring to code a calculator, all the same. Unless I start with addition and subtraction only…

A pet project of mine would be to encode a difference engine. So it will ideally go through a sequence of numbers up to my desired power.

Division in Python 3 always produces a float. Int division produces an int. If not otherwise stated, the array produced from range() always starts with 0, and 0%x = 0 for any x.

When traversing a string for char, keep in mind that the case is very important. Uppercase and lowercase letters are seen as distinct.

When using loops, must make sure to 1) Initialise the variable to avoid NameError, and also prevent using a variable already in use elsewhere which may otherwise introduce unexpected variation; 2) change the variable within the loop to avoid infinite loops.

\*\*Procedure: Set the variable, Write the loop, Decrement variable within loop, Check variable

Guess-and-check aka Exhaustive Enumeration

Made a ‘simple’ VowelCounter function. The actual calculation was easy, but getting the indentation, print and calling of the function was the hard part. Apparently I have to import the function from its file. Kept using the wrong term like ‘load’ or ‘call’.

Now I made a bobCounter. Sadly I forgot about len() so I tried to redo it, but actually a for-loop was good enough.

Difference engine:

1. Initialise array of range(11)
2. Raise each entry to the user-inputted value as power. Cast the input as int
3. Find the difference between each entry.
4. Repeat the difference and keep count of how many levels of differences there are.

Exceptions

* IndexError
* TypeError
* NameError
* SyntaxError
* AttributeError
* ValueError
* IOError

Responses to exceptions

* Fail silently – Very bad
* Return an error value – Need to find which value, adds complication
* Stop execution to raise exception

Iteration: for and while loops, use of if and range.

Recursion: Calling the same function within itself, until it reaches a base case. So it goes down and then goes up.

Recursion might be more intuitive, more efficient from the programmer’s POV. But it might not be so efficient from the computer’s POV. Analagous to mathematical induction. More than one base case may also be required when the function is recursed multiple times.

Towers of Hanoi is a problem that is hard to solve through iteration, yet has an elegant solution through recursion. Need to break it down into a smaller version of the same problem – moving stacks of smaller and smaller sizes. Then assumed those smaller versions can be solved and combine these smaller cases with other operations.

List comprehensions and compressing code into fewest lines.

yield(), generators, iterables.  
Iterables are lists generated via list comprehension, using []. Their values are stored in memory.  
Generators are similar to lists but use (). This means that their values aren’t stored in memory, but only *generated* when required  
Yield() allows for a function to return a generator object, without actually running its own code. The code will only run and return its values *when required*.

Modules as files containing different functions… Much like what I’ve been doing!

Mutation, Aliasing and Cloning – Being aware of the relationship between an object and the data it holds, notably when one object is defined in terms on another. This can lead to unexpected changes that cascades across variables. A solution is to create a temporary copy of the variable in question so that any changes made to it won’t prematurely affect how it’s handled by the function.  
Seems to apply for lists and not tuples (which include strings).  
Pointers will be the same for identical objects, but different for objects that contain the same data only (like two lists with the same data entries but declared separately).

Higher level programming – Coupling functions with lists. In fact, functions are first-class objects because:

* They have types
* They can be elements of data structures like lists
* They can appear in expressions (either as part of an assignment statement or a function argument)

Lambda functions resemble expressions in list comprehension. They are one-time functions that are used only within a smaller scope. They get declared and used, and are limited to within the parenthesis.

Dict example: Oddity is that the frequencies tuple (beatles) seems to have been changed during the demo and that’s why the most common words were different from what could be observed in the list.  
It seems to be more profound than I can understand right now. Being able to iterate over the data and manipulate it, calling each key as a word and so on.  
I just made the connection. This topic will be important for understand OreDictionary, wouldn’t it?

Dictionaries can be used for memoisation as well. Allows storage of previous outputs that would be useful later on, avoiding the need for re-computing. The dictionary is appended to as the function proceeds e.g. for Fibonacci sequence.

Use of global variables to count how many times a function is called. The number of calls can be a measure of efficiency.

**How to Debug**

Analogy of bugs in soup: Block them out, scoop them out or clean the kitchen.

DP 🡪 Test, DP 🡪 Debug

1. Defensive Programming
   1. Write specifications for functions – docstrings
   2. Modularise programs vs writing a long function.  
      Check each piece instead of the whole program.
   3. Check conditions on inputs/outputs (assertions)
2. Testing/Validation
   1. Compare input/output pairs to specs
   2. Find out when it doesn’t work
   3. Try to break the program
3. Debugging
   1. Study events leading up to an error
   2. Why isn’t it working?
   3. How can I fix the program?

Start writing code that is testable and debuggable

1. Motherhood approach: List **all** assumptions, even the universal or obvious ones. Serves as good practice for other assumptions as well

Ensure code runs, and have a set of expected outputs (may want to type this out elsewhere for each input set).

Classes of tests:

1. Unit testing: Test each function separately
2. Regression testing: Add tests for bugs as you find them in a program to catch reintroduced errors that arise from added code i.e. fixes.
3. Integration testing: For the overall program

Testing approaches

1. Intuition about natural boundaries to problem
2. Random testing: Provide larger data set as input
3. Black box testing: Explore paths through specs  
   Based on the specs, what kind of inputs can go into the program?  
   What boundaries, test cases, or extremes are there?
4. Glass box testing: Explore paths through code  
   Ideally path-complete
   1. Branches: Exercise all paths of a conditional
   2. For loops: Loop not entered, loop executed once, loop executed more than once.
   3. While loops: Same as for loops, and cases that catch all ways to exit the loop.

Brain is frying from glass box testing. Arguably more tedious than black box testing if you confuse them. Yet if you really follow through the code, then glass box would be more straightforward. I guess. You might miss one of the conditions in black box. Well, both are hard, that much is clear.

Responses to bugs:

1. Isolate the bug
2. Eradicate the bug
3. Retest until code runs correctly

Runtime bugs:

* Overt vs Covert  
  Overt – Obvious manifestation. Code crashes or runs infinitely.  
  Covert – Not obvious. Value may be wrong.
* Persistent vs Intermittent  
  Persistent – Occurs every time code is run  
  Intermittent – Only occurs sometimes even with same input
* Overt and persistent – Easy to detect and can use defensive programming against it
* Overt and intermittent – Annoying but can be handled if the conditions to trigger it can be reproduced
* Covert – The most dangerous because it can produce incorrect/illegal values

Use multiple print statements to analyse working values. Can combine with bisection search

Study the code

Use the scientific method

Practise version control

Logic errors are harder to handle. Need to think clearly about what the code is doing and explain it.

Use try-except clauses to manipulate data further

Raise keyword allows customisation of error messages to provide more data on what went wrong e.g. function with incorrect argument

Assertions as defensive programming

* (Don’t allow controlled response to unexpected situations)
* Ensure that execution halts when an expected condition is not met
* Typically used to check inputs to function procedures but can be used elsewhere
* Can be used to check outputs of functions to avoid propagating bad values
* Can ease location of the source of a bug

Where to use assertions  
Goal: Spot where bugs have been introduced and mark where they happened

* Supplement to testing
* Raise exceptions in response to bad data input
* Check types of arguments or values
* Check that invariants on data structures are met
* Check constraints on return values
* Check for violations of constraints on procedure

(I have a tendency to use if-else over try-except)

Object-Oriented Programming

Each kind of data is an instance of an object. Each object has a type, an internal data representation (primitive/composite) and a set of procedures for interaction with that object.

Objects are a data abstraction that capture:

* Internal representation through data attributes
* Interface for interacting with object through methods (procedures), defines behaviours but hides implementation.

Distinction between creating a class and using an instance of that class (for some reason I’m thinking in RPG terms)

Creating the class involves:

1. Defining the class name
2. Defining class attributes

Using the class involves:

1. Creating new instances of the objects
2. Running operations on the instances

Advantages

1. Bundle data into packages – Abstraction
2. Divide-and-conquer development
3. Classes make it easy to reuse code

Crafting Recipe Exporter

I have a pretty clear idea of what it would look like: JEI/NEI outside of a Minecraft instance. So with a large search page, and a single active tab at least. That’s the basic. Other things that would be useful would be saved/favourited recipes, a calculator, the crafting implement tab, InpureCore-style filtering of redundant entries, sorting by mod/name/difficulty and maybe Crafttweaker integration.

However, the basic version is actually a lot more involved than it appears. It seems that it dynamically generates the recipe and ingredient data based on the JSON files. These JSONs aren’t the most readable on their own, since each recipe takes up a single line because technically it’s an array. Apparently CraftTweaker can export recipes, but cannot tell which mod contributes which recipe.

These are the tasks that JEI/NEI handles when bringing up the recipes:

1. Read the JSON files
2. Sort them by mod
3. Sort them by name
4. Convert the JSON array into a 3x3 image
5. Add a hyperlink to every item, whether as a recipe or an ingredient

Likely the mod will have to run once during game time so as to export the entire file. Then this file will be read by my own app that will reconstruct it.

There’s actually somebody who tried doing it: AndrewMiner, the Crafting Guide person. But so far what he has seems to be only a website. His mod only works for 1.6.4. Hmm, maybe I should create a new instance and test it. No, I was wrong. It is used in conjunction with a website. In other words, the data it exports is readable by the website, rather than another app. Still I should study it.

It seems this project will have two parts: a mod for exporting the recipes, and an external app that recreates the JEI/NEI experience. The simplest way would be to start with vanilla recipes. The sad thing is that there’s just not much documentation on what JEI/NEI do or how they work. For example, I don’t know how they interact with the new crafting tab in vanilla, or whether they plan to at all.

Other mods:

* TMI for 1.6.4
* NEI for 1.7.10
* JEI for 1.8+
* Just Enough Calculation
* CraftTweaker for 1.7.10 and 1.8+
* MineTweaker
* RecipeDumper for 1.6.4
* 1.12 Recipe book

Slimycraft

Adds a lot more to do with slimes to the game. New items, new mobs, new worldgen, new blocks, new crafting. Maybe even new mechanics and a new dimension.

(Slimecraft is the name of a server that doesn’t actually do a lot with slime)

Slimes in vanilla: Spawn either in swamps at night depending on moon phase, or at any light level below y = 32 in a slime chunk. They don’t spawn in Peaceful mode.

Modules

* Décor and Building
* Mob farming (Smile Rancher)
* Mobility
* Gourmet and Brewing
* Armory and Arsenal
* Fluidics and Piping, Plastics
* Worldgen and Dimension
* Slime Infusion of mobs and blocks

Slimeball derivatives (Gourmet)

* Core
  + Nata de coco
  + Snow globe
* Paste
  + Sugar
  + Tofu
  + Jelly
  + Syrup
  + Preserves
  + Stew
* Film
  + Bubbles
  + Boba
  + Strips
  + String/Thread/Fiber
  + Skin/Leather
* Presses
  + Pancake
  + Juice
  + Jerky
* Crystal
* Ingot
* Dust

Décor and Building

* Slimes of colour
* Blocks, Bricks, Bullion
* Fluids

Refinery and Fluidics

* Cauldron/Vat
* Distillation Tower
* Curing vat
* Tubes
* Droplet generator
* Plastics
* Plexiglass
* Paints
* Fountain
* Pump
* Mould
* Golem

Armory and Arsenal

* Slimesuit
* Pellets
* Sword with Knockback II
* Axe with Knockback III
* Boxing gloves with Knockback IV, no damage
* Bow with Punch
* Glue
* Spray

Worldgen and dimensions

* Slime Island
* Slimordial soup
* Portal

Save Game

This would be based on backups, which are compressed save files. Now I recognise that it simply extends the idea of backups. If you want to save your game, then you’ll initiate the backup and wait for it to complete. Autosaves happen well, because of backups. Loading happens when you extract a backup and transfer it to the saves folder.

What would my mod offer above a backup mod? Well, maybe a totally ingame experience. Saves a few minutes at most. Then perhaps it lets you save and load a game at any time. The issue is that unlike other games, Minecraft doesn’t normally produce its own save files. No, that’s not the issue. The issue is that the save files are so huge. Maybe I won’t worry about that.

OK, I think I get it. In other games, your current world is lost when you load a save. However, in Minecraft the current world remains. Now this is how Minecraft works: Think of it like containers and clones. In normal games, the main container – the one that the player uses – has a clone of one of the saves. However, the clone is lost when you load back a save. In other words, at any one time there will only be one saved clone. But in Minecraft, as soon as a save is loaded into the main container, that ‘main’ clone is saved as well, so that there will be two clones with the same origin.

In other words, making this single switch will fundamentally change how Minecraft works. Now why can’t I just accept it? Well, this will result in a constant increase in the number of saves every time you load a previous save. The ‘main’ world will be automatically saved and then kept in the ‘saves’ folder. And when you do this several times, this results in several ‘main’ saves.

Now sure, people might like it. But it’s dangerous because it’s challenges one of the basic assumptions when you quit a game: That the game will save on exit. Now a way around this is to offer an option when loading a save: To delete the current world or keep it. This preserves the save-on-exit functionality, though it makes loading saves slightly more difficult. I doubt that anyone would want to keep the ‘main’ save though. If they do, I’d need them to rename it.

Anyway, the load menu would resemble the usual world screen. But I suppose it combines the tasks of closing the ‘main’ world and loading the new save. Could be an issue on weaker computers or larger saves, since closing and opening worlds actually can take a few minutes.

Others

A way to optimise a modpack. One that you wouldn’t be adding mods to any longer. Or updating.

Forge

Forge is an API that is built on top of the base Java code of Minecraft. It is a mod loader that serves to abstract away the lower-level code in favour of a more universal and scalable interface. It is crucial for allowing mods to work together with each other, allowing the assembly of modpacks.

Hooks

Event handling

4096 IDs, unlimited sprites

An **application programming interface**provides all the building blocks for a program along with methods of communication between them.

(Even for a standard as fundamental as Forge for the modded Minecraft community, there’s not much of a description on what it does.)

Java (Eclipse) 🡪 Forge 🡪 Minecraft