CALCULATOR

10/10/18

Sharnbrook Academy (15163)

James Wilson (5927)

Contents

Analysis

Introduction

I will make a general-purpose scientific calculator.

I chose to make a calculator because it combines computing with maths – modelling mathematical operations in code. Once finished, it will be a general-purpose scientific calculator, like handheld ones but on the computer. This means it can not only be used by users but be called by external programs to calculate answers to mathematical expressions they may need an answer to. This makes it an abstraction other programs can benefit from.

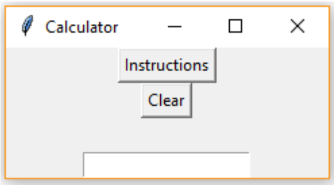
For example, if a programmer wanted to find the solution to a quadratic equation in sine, they will be able to use my calculator to find the solutions to the quadratic and then use my inverse sine function to find the acute angle for each solution of the quadratic. From this point, they can use the addition, subtraction and multiplication operators and pi constant to find all other solutions in the desired range.

To determine what features my calculator should have, I need to consider what the users will want and what existing calculators offer.

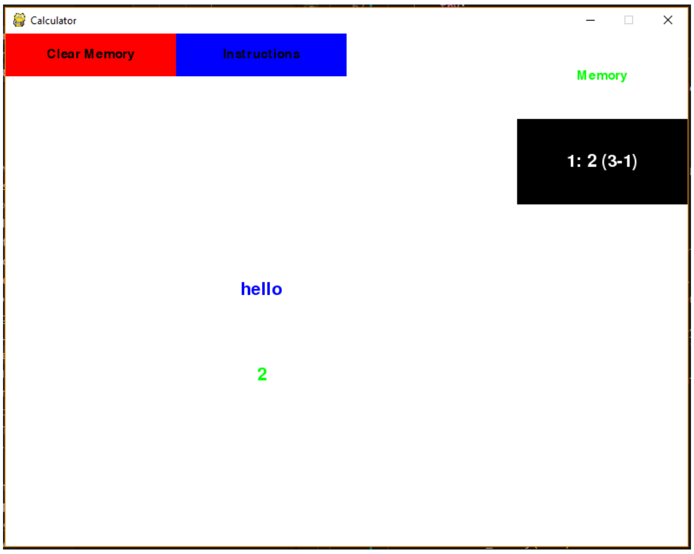
Prototype GUI Design

I prototyped a GUI design in both Pygame and Tkinter to ask potential users which they preferred. Pygame can be more colourful but it takes more effort to format things correctly whereas Tkinter is often more formal in fields, etc. but this means the text can't overlap other things like it can in Pygame. These prototypes are simply windows with a few buttons that don't work to ask the users which they prefer.

Tkinter



Pygame



Potential Users and Interviews

To determine which features my calculator should have and how the interface should look, I have thought of 3 potential users of my calculator and interviewed them:

1) My friend is writing a program to model the trajectories of projectiles, so he needs a way to work out the values of expressions in order to display the height of the object at each time. This is not too hard when the equations are fixed but he wants the users to be able to input expressions which need to be parsed and executed which is much harder. This makes him the perfect user for my calculator – he can use his own interface to get the expression from the user and use the answer, but he can call my calculator to find the value of the expression quickly and easily at different input values. For him, it is an abstraction where he does not need to know how it works, just how to use it.
2) My maths class often use quadratic equation solvers on our calculators so that we can solve quadratics along with the basic operations, exponentiation and factorial. We also use nCr to find the combinations for binomial expansion. This is a general user – no specific reason to use the calculator but using various features regularly for various reasons.
3) Our computing class often uses the mod operation and conversions between denary, binary and hexadecimal.

Interview Questions

1) What features would you like a calculator to have? What features do you often use on a calculator?
2) What obscure features would you like a calculator to have? What features are very difficult or time consuming to do by hand so would be good in a calculator regardless of how commonly it is needed?
3) How would you like a calculator on the computer to look:
   a) Would you like buttons for characters or just use the keyboard?
   b) Would you like to be able to view the memory at the same time as using the calculator normally?
   c) Would you like the instructions to be accessible on the calculator or in a separate instructions book?
   d) Any other thoughts on how you would like the calculator to look?
4) Which modes of the grey calculator do you normally use?
5) Which would you prefer the calculator to look like?

Question 4 is to inform my choices of modes in my calculator which I will discuss in the case study.

Question 5 is to inform my choice of library to help me display the window.

Interview 1 – my friend making the projectile modelling software

1) The ability to solve complex calculations involving multiple functions, such as trigonometric functions and factorial. Often, I use the basic operations to solve questions in physics and maths.
2) A quadratic equation solver and a statistics tool to find statistics about data like the mean and variance.
3)
   a) Buttons
   b) Yes
   c) On the calculator
   d) I would like it if the calculator had the basic template that follows standard hand-held calculators. I think this would make it an easy transition between the two.
4) Almost always the normal mode but sometimes the STATS mode.
5) Pygame - bigger and colourful buttons and text will make it nicer to use

Interview 2 – a student from my maths class

1) I most often use addition, subtraction, multiplication and division so it should definitely have these.
2) Finding the LCM and HCF of 2 numbers. We also often need the combinations and permutations formulae for binomial expansion so an nCr and nPr operation would really speed things up.
3)
   a) Keyboard is fine if it will always be on the computer
   b) Yes, that would be very useful to look back on pervious answers
   c) Yes, much better than having a booklet but as long as the instructions are easy to find in the interface!
   d) A big main screen with lots of space to display what you are inputting and the answer.

4) Always COMP.
5) Pygame - a simpler and more colourful design

Interview 3 – a student from my computing class

1) I mostly use the normal 4 operations and powers
2) In computing we often need to use mod. You can do this with division, subtraction and multiplication but it is much longer than having a specific operation.
3)
   a) Keyboard
   b) Yes
   c) Yes
   d) Simplistic design so it is clear
4) Normal mode
5) Pygame - clearer

Interview Analysis

Operations

The operations my interviewees requested are:

- Addition
- Subtraction
- Multiplication
- Division
- Exponentiation
- Mod (remainder after division)
- Factorial
- Combination
- Permutations
- Trigonometric functions
- LCM (lowest common multiple)
- HCF (highest common factor)
- Quadratic equation solver
- Statistic tool

I plan to include all of these apart from the statistics tool because there can be a variable number of data points. In order to do this, I would need a separate mode in the calculator, and this will be time consuming for not much use. If I had more time, I would add this in, but it will require too much work for too little gain.

GUI Design

I got a 2/3 response to use keyboard keys rather than on screen buttons to input expressions. Interviewee 2's answer to question 3d was a request for the screen to be big and buttons would crowd it so I will use keyboard keys.

All interviewees thought that being able to read instructions on the calculator and view memory while using the calculator normally are a good idea so I will do both of these.

Interviewee 1 said that the calculator should look like handheld calculators which fits with him wanting buttons, but the other 2 just wanted a simplistic and clear design with no buttons. I will make the calculator as clear as possible with as little as possible cluttering the display.

How Existing Calculators Work

Token Types

Components of expressions are called tokens so the number '13.1' is a token, the operator '+' is a token, etc. The calculator will need to be able to deal with the following types of tokens:

- **Number**: a numerical value – whole number or decimal which could be in standard form or a word which represents a constant (e.g.: pi meaning 3.1415…)
- **Brackets (open and close):** Although treated separately, they still need to be identified and processed
- **Operator**: a word/symbol that performs an operation:
    - **Unary**: takes 1 operand – on either the left or right side
    - **Binary**: takes 2 operands – 1 on each side
- **Function:** A word followed by brackets with operands in separated by commas

Order of Operations

In order to be executed correctly, following BODMAS, every operation needs a precedence and associativity which defines the order in which they are executed.

Precedence

In the following table of precedence, lower precedence means executed first.

Precedence	Description
1	Exponentiation
2	Unary operators
3	Multiplication, division, mod, div
4	Addition, subtraction

Table 1: Operator Precedence

Although not in the above table, brackets always have the highest precedence and functions must be executed to get a number before that number can be operated on so both of these come before all those in this table.

Associativity

Operators also have associativity to determine which is executed first if they have the same precedence. Associativity can either be left-to-right or right-to-left and most operators are left-to-right associative.

Left-to-right associativity means the left-most operand has to be unambiguous (not involved in another sub-expression) for that operation to be executed first. This means the left-most operator is executed first and vice versa for right-to-left.

Unary operators can be prefix (the operator goes before the operand) or postfix (the operator goes after the operand). Prefix operators are usually right-associative and postfix operators are usually left-associative[1] and as there don't seem to be any examples of operators which contradict this[2], I will assume this is true in all cases. Therefore, within this calculator, the associativity of unary operators will be the side of the operand in relation to the operator.

Existing Software – Grey Calculator

In order to identify weaknesses of current software and see where I can improve on this in my calculator, I will look at the grey CASIO fx-85GT PLUS which is a common entry-level scientific calculator.

Modes

The calculator has 4 modes:

[1] The first few lines under the 'Associativity' heading in http://www.cs.man.ac.uk/~pjj/cs211/ho/node2.html
[2] https://stackoverflow.com/questions/14084421/is-there-such-thing-as-a-left-associative-prefix-operator-or-right-associative-p

- COMP – the main mode for most operations
- STAT – a statistics mode which allows you to find summary statistics about data such as the mean and variance
- TABLE – allows you to input an expression in terms of x and gives the value of that function for a range of values
- VERIF – allows you to input 2 expressions in terms of x and tells you whether or not they are equal for all values of x

In my experience and from asking my interviewees, the COMP mode is the only commonly used mode so I will only have this mode in my calculator.

Features

In COMP mode, the calculator has many features:

- Operators:
    - Addition
    - Subtraction
    - Multiplication
    - Division
    - Abs
    - Exponentiation
    - Roots
    - Combinations
    - Permutations
- Functions:
    - Logs including natural log
    - Circular functions and their inverses
    - Hyperbolic functions and their inverses
- Memory storage and recall
- Constants:
    - pi
    - e
- Random number generation
- Standard form

General

The grey calculator comes with an instructions book which is very long and not very easy to read.

You can change settings such as angular unit on the calculator, but you have to navigate a menu by pressing numbers corresponding to options which is time consuming and not very user-friendly.

It is complex to save and recall memory and you can only save 6 answers in memory, not including the most recent 'ans'.

Error messages are simply a blank screen with the type of error that occurred:

- 'syntax error' if the expression doesn't make sense, e.g.: having more close brackets than open brackets.
- 'math error' if the expression lead to something that can't be calculated such as division by 0.

Grey Calculator Analysis

I can include all operators and functions from the grey calculator in my calculator and I will include memory functionality. My calculator will have constants including pi and e and be able to understand and output in standard form. I can add a function to generate random numbers too.

I noticed some weaknesses of the grey calculator so I will plan to improve on these in my calculator in the following ways:

- My calculator will have a large, clear GUI which can explain things better, helping in the following areas:
    - **Instructions**: rather than having to supply an instructions book, I will be able to display instructions on the screen, and these can be less detailed than those of a handheld calculator as it is more intuitive in other ways, e.g.: being able to click on buttons to navigate the interface as opposed to having to press a key that corresponds to it.
    - **Memory**: I will display the most recent answers in memory to the side of the expression and answer area so users can see their previous answers simultaneously to using the calculator. They will also be able to click on a memory item to insert that answer into the expression.
    - **Errors**: Where traditional calculators often have a separate screen showing 'Math error!' with no more information, my calculator will display an intuitive error message simultaneously to the user using the calculator normally.
    - **Settings**: Traditional calculators have settings, but it will not be as convenient to view and change them as it will be in my calculator.
- Due to the calculator being on a computer rather than only an isolated hand-held device:
    - The speed of calculations will depend on the hardware of the computer so if better speeds are required, it is possible to upgrade hardware whereas on handheld calculators, it is fixed.
    - The calculator will be accessible in many different ways – not only in the GUI like handheld calculators but advanced users will have access to a command-line interface, programmers will be able to add custom user interfaces or use it to calculate answers to be used in other programs. This is explained in more detail in the Entry Points and User Experience section.
    - It has the potential to be turned into an app, so it is accessible on any device and always with you
- Traditional handheld calculators only calculate answers to 10 significant figures, but this calculator will be able to work at a much higher accuracy (up to 28 decimal places). This calculator will be able to round answers to the desired number of places.
- It will be trivial to add more operations to the calculator so it can scale very easily. Any function written in any programming language will be easily representable in the calculator with any number of parameters.

Objectives

A calculator is quite broad, and the scope could be anything from only add, subtract, multiply and divide to every operation you could imagine. To decide on the scope of my calculator, I have looked at an existing calculator and asked potential users. In doing this, I have decided on the following objectives for my calculator:

1) Must be able to perform all of the following operations correctly for all valid inputs and produce an intuitive error message for all invalid inputs (must never crash):
    a) Addition
    b) Subtraction
    c) Multiplication
    d) Division
    e) Exponentiation
    f) Roots
    g) Logs including natural log
    h) Mod (remainder after division)
    i) Factorial
    j) Absolute value
    k) LCM (lowest common multiple)
    l) HCF (highest common factor)
    m) Combinations

n) Permutations
o) Quadratic equation solver
p) Circular functions and their inverses
q) Hyperbolic functions and their inverses
r) Random number generation

2) Error messages get an average rating of at least 7/10 from my 3 interviewees for being intuitive
3) Must automatically store all answers in memory and allow them to be recalled
4) Must be able to translate the following constants into numbers stored accurately:
   a) pi
   b) e
   c) And any others I add
5) The output should change correctly depending on settings such as how to round and whether angles are in radians or degrees
6) Has a GUI which gets an average rating of at least 7/10 from my 3 interviewees in the following areas:
   a) Clarity of instructions
   b) Intuitiveness of navigation
   c) Entering expressions
   d) Reading answers
   e) Reading error messages
   f) Interacting with memory
   g) Changing settings

Decomposition into Stages

I have split the calculator into 8 stages, so I build the calculator from the ground up. If I wrote all the code and then found lots of errors when testing, it would be very difficult to work out which part the errors are from and fix them. This way, I will write a small bit of code and test it to make sure it is correct before moving on to the next part. This means, I will design, code and test each stage before moving onto the next stage.

1) Core Functionality

Firstly, the calculator needs to be able to take an expression and produce an answer. But to test this is working correctly, I will include each of the following types of token to test the core functionality:

- **Exponentiation:** right associative binary operator with precedence 2
- **Subtraction:** left associative binary operator with precedence 4
- **Negation:** right associative unary operator
- **Factorial:** left associative unary operator

A range of these is needed as I need to test that it can distinguish between unary and binary operators, so I need a binary operator with the same symbol as a unary operator – subtraction and negation. I also need 2 binary operators with different precedences and associativities – exponentiation and subtraction. And finally, 2 unary operators with different associativities – negation and factorial.

At the end of this stage, the calculator should pass all tests – not necessarily be able to work for all expressions (as it only knows about the 4 testing operations) but not to crash and to work correctly for these testing operations.

2) Interface (Memory)

Before I can create a user interface, I want the calculator to be able to store previous answers. Therefore, this is not a *user* interface but the interface between the user interface and the calculator.

External programs will only import this if they are creating their own user interface (and want to use my memory system), they will not if the calculator is going to be used by a program (as the program can record its previous calculations itself if it needs to).

This will not analyse the expressions passed to it, but simply store previous answers in memory and allow them to be accessed.

The requirements for this stage are:

1) A user interface should be able to call the calculator via this
2) A user interface should be able to retrieve the entire memory or a specific entry
3) A user interface should be able to clear the memory

I will also provide a range of documentation for users and programmers – there will be instructions for users and docstrings on all files, classes and subroutines for the programmers.

3) Graphical User Interface

Now I need to make the calculator as user-friendly as possible by adding a graphical user interface. This will facilitate normal use, memory and instructions.

Lots of calculators are not very user friendly as it is unclear how to do things – you have to read pages and pages of instructions to know how to retrieve the memory. Therefore, I aim to make this calculator much simpler to use with buttons for these functions as well as the keyboard for most symbols.

I need to provide a way to:

- Input expressions into the calculator
- View the answers to these expressions
- View memory
- Insert previous answers into expressions
- Clear the memory
- View instructions

4) Constants

The calculator needs to be able to deal with constants such as ‘pi’ and ‘e’ so users don’t have to type them out every time and they can be stored very accurately to give the best answer.

The calculator should be able to display the exact decimal form of constants such as ‘pi’ and ‘e’ and allow them to be used in expressions.

5) Standard Form

Very big and very small numbers are difficult to represent exactly without standard form so every calculator should have it.

The calculator should be able to understand numbers in standard form and it should output numbers in standard form if they are very big or very small. It should be able to use numbers in standard form in expressions.

6) Functions

At the moment, the calculator can only deal with operators, not functions.

There is an argument that functions, as I define them, are not needed in calculators. This is because most mathematical functions take either 1 or 2 operands, so their syntax can be changed from using brackets and commas to the same as operators. For example, the sine trigonometric function which is normally expressed with brackets in function notation, e.g.: ‘sin(30)’, can be expressed as a right associative unary operator in operator notation, e.g.: ‘sin 30’. However, functions give the calculator the freedom to have more than 2 operands and this means any coded function can be modelled in the calculator.

I will include the testing functions:

- sin - this will be in the final calculator
- root - this will be an operator in the final calculator but is only a function for testing
- sum - this will not be in the final calculator and is only here for testing

At the end of this stage, the calculator should be able to deal with functions with any number of parameters.

7) Operations

By this stage, I will have the groundwork of the calculator complete so I can now add more operations to make the calculator much more useful.

The operators in yellow I will have already added to test other features, but I may change them slightly.

Binary Operators

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Floor division (\)
- Remainder after division by modular arithmetic (%)
- Exponentiation (^)
- Root (¬)
- Combinations (C)
- Permutations (P)

Unary Operators

- Positive (+)
- Negative (-)
- Factorial (!)

Functions

- Natural log (ln)
- Logarithm (log)
- Modulus AKA absolute value (abs)
- Lowest Common Multiple (LCM)
- Highest Common Factor (HCF)
- Random number generator (rand)
- Quadratic equation solver (quad)
- Circular functions:
    - Sine (sin)
    - Cosine (cos)
    - Tangent (tan)
- Inverse circular functions:
    - Inverse sine (arsin)
    - Inverse cosine (arcos)
    - Inverse tangent (artan)
- Hyperbolic functions:
    - Hyperbolic sine (sinh)
    - Hyperbolic cosine (cosh)
    - Hyperbolic tangent (tanh)
- Inverse hyperbolic functions:

- Inverse hyperbolic sine (arsinh)
- Inverse hyperbolic cosine (arcosh)
- Inverse hyperbolic tangent (artanh)

Constants

- pi = 3.14159265358979323846264338327950288
- tau = 6.28318530717958647692528676655900576
- e = 2.71828182845904523536028747135266249
- g = 9.80665
- phi = 1.61803398874989484820458683436563811

8) Settings

To make the calculator more usable, I will add settings which can be changed in the interfaces or passed by external programs. This can contain settings such as whether or not to round, how many decimal places to round to and the angular unit to use.

Entry Points and User Experience

As discussed above, I want the calculator to be used by both actual users and programmers. This means there will be 4 entry points to my calculator. For each entry point, I will discuss what it is, who it is for, and how I can make it as user friendly as possible. Users will not use the calculator if it is difficult or awkward, so I will use the user interviews I did at the start of analysis to inform my design of the user interface.

Core calculator for external programs

Programmers will be able to import the calculator's core functionality to their programs and call a function with their expression to return the answer.

To make it as easy as possible for programmers to use, I will include docstrings for the relevant functions which explain all parameters and return values. The programmer will also be able to provide settings such as rounding method and accuracy with the expression to personalise the output. I will also provide a README which explains this.

Interface for custom made user interfaces

Programmers will be able to import the interface to use the calculator's core functionality and memory. They could then create a custom user interface but use my calculator's core functionality and memory in it.

Again, docstrings will be provided as well as a README.

Command-line interface for users

I will make 2 command-line interfaces for testing (1 for the main calculator in stage 1 and 1 for the memory interface in stage 2) and I will leave them in for users who might want them. Some power-users who are experienced with using my calculator, know what they are doing and want to do things fast (a command-line uses less computer resources than the GUI) may want to use these. The CLIs will not be as user friendly as the GUI but I will do as much as I can to make them intuitive without sacrificing speed.

Graphical user interface for users

Users will also have access to the GUI which will display a window much like many other programs to interact with. This is easier to use and better for users new to my calculator as it is less intimidating than a command line as buttons clearly show some functionality. I will design the GUI to be as user friendly as possible in the documented design section.

Limitations

There are 2 main limitations of this project:

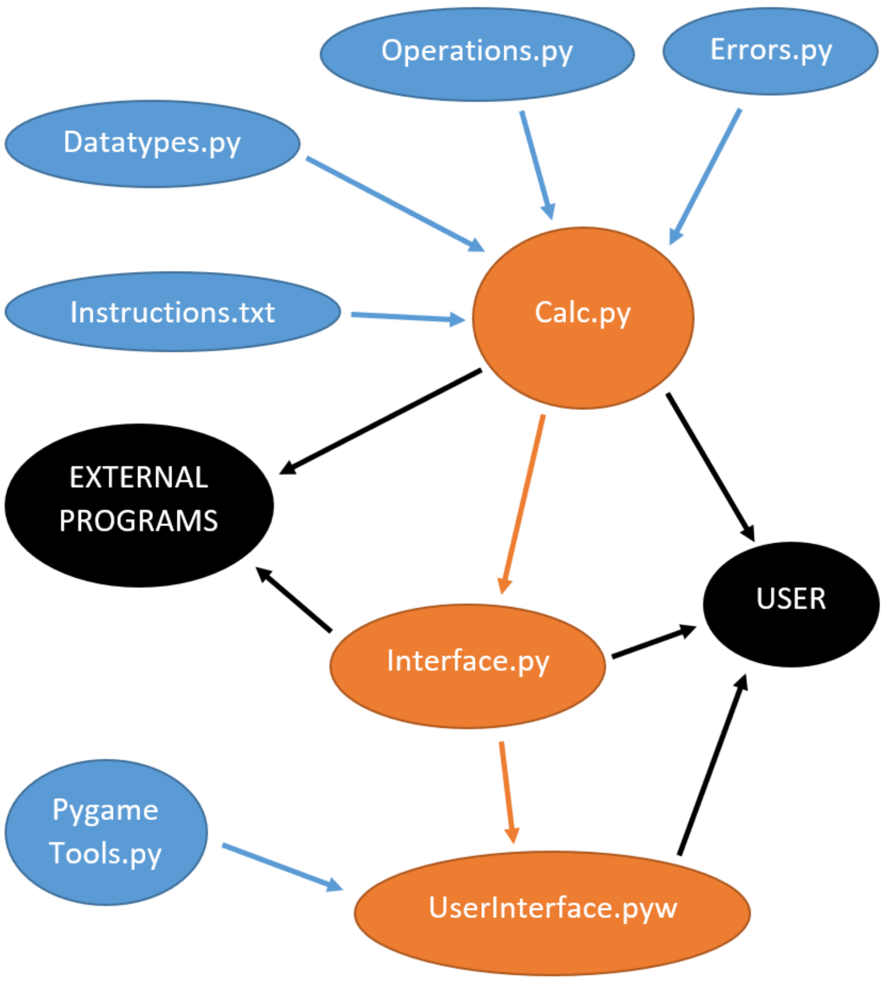
- There are too many mathematical operations to model in this calculator, so it will not suit everyone straight away. However, I will include the main operations, so it is suitable in most cases and I will make it simple to add new operations.
- The GUI design is subjective so will never be perfect for everyone, however I will allow other programmers to import the calculator's core functionality but make their own user interface, so they can improve on it if they want.

Documented Design

Files, File Interaction and Entry Points

To keep the code as clean as possible, I will split it into different files which interact with each other in various ways. Not all files will be created until the end. The file names are in bold:

- Orange: Main files for operation:
    - **Calc.py**: Calculator's core functionality and settings.
    - **Interface.py**: Interface between user interfaces and the calculator and implement memory.
    - **UserInterface.pyw**: Graphical user interface.
- Blue: Helper files:
    - **Errors.py**: Contains custom errors for use in the calculator to tell the user what they are doing wrong.
    - **Datatypes.py**: Contains datatypes to be used in the core calculator such as stacks, queues, functions and operators. These are not operations users can use in the calculator, but they are used internally for parsing the expressions.
    - **Operations.py**: These are operations that can be used within the calculator (both operators and functions). More can be added trivially.
    - **PygameTools.py:** Helper constants, functions and classes for displaying text and buttons in Pygame.
    - **Instructions.txt:** Contains the instructions for all parts of the calculator.
- Black: Endpoints (not files but people/things that will use the calculator):
    - **EXTERNAL PROGRAMS**: Other programs can call the calculator for single calculations with or without memory.
    - **USER**: The user can use the calculator with memory through the user interface or use one of the command line interfaces I made for testing purposes.



IPSO Chart

Input	Process	Storage	Output
Expression	Calculate answer (to be broken down further later)	Save answer in memory	Answer
Request to view instructions			Instructions
Request to clear memory	Clear memory	Clear memory	

Table 2: IPSO Chart

1) Core Functionality

In order to turn an expression into an answer, the calculator needs to do 3 things: 'tokenise', 'convert' and 'execute' (and then 'calculate' links them together). I will write each of these as subroutines and explain and write pseudocode for them below.

I considered implementing this recursively – if the calculator was called with more than 1 token, it would work out which operation to execute first, execute that and insert the answer back into the expression and call itself again with this new expression. However, it is trivial to convert this into an iterative algorithm and recursive algorithms use much more memory as they have to store all local variables on the call stack and create a whole new set of them for each step deeper. For this reason, I do it iteratively, but in stage 2 I use recursion to execute the operands of functions.

Errors

There are 2 types of errors that could occur in my calculator:

1) Type 1: Errors within the expression – the user's fault, not a problem with the code, e.g.: a missing operand
2) Type 2: Errors within the code – the programmer's fault, e.g.: a reference to an undefined variable

Obviously, I don't want the program to crash on the user because they will not know what has happened. However, I don't want a type 2 error to be presented to the user, e.g.: "undefined variable 's'", as they will not know what it means, and this makes it harder to realise it is type 2 error that needs to be fixed. This means I need to handle these errors differently – type 1 errors should be handled and given as a meaningful error message to the user, whereas type 2 errors should be raised as normal exceptions so it is as clear as possible to testers what caused the error so it can be fixed.

Therefore, type 2 errors are handled as normal but how do I handle type 1 errors? There are 3 ways I considered:

Option 1: Return an error flag

Instead of returning just the answer from functions, also return a second Boolean variable that will represent whether or not there has been an error. If there has not been an error, the first return will be the answer expected however if there has been an error, the first return will be the error message to give to the user. When receiving these from a function, the error flag should be checked first and if true, it should return immediately until the interface gets it and displays it to the user.

This seems like a reasonable idea however the first return variable changes types – from a number if there is no error to a string error message if there is. Python allows this but other languages do not so this is not good programming practice. Also, lots more code would have to be written to check whether an error has occurred and stop everything else as it has. This seems unnecessary as a normal exception, unless it is caught, will do this for me.

Option 2: Raise and catch a custom error

Type 2 errors will raise existing exceptions depending on the nature of the error, but I could create a new exception representing a type 2 error, e.g.: 'CalcError'. Then when calling code that may have a type 2 error, I could 'try' it and catch this specific error which will only catch type 1 errors, and this will hold the error message within it.

I will be raising an exception only to catch it again, but I think this is best because it is an easy way to store the error message and the programming language will handle exiting the program until I catch the error.

Datatypes

I will need a few custom-written datatypes for the calculator – some to identify types of tokens and others to store tokens when executing expressions.

I will also need a dictionary to store valid tokens (binary operators, unary operators and functions) that could be in the expression. The key will be the symbol in the expression and the corresponding value will be an object of one of the classes below to identify types of tokens.

To identify types of tokens

Different types of tokens need to be treated differently so I need to identify them by making them instances of these classes so I can easily check what type of token they are when I need to decide what to do with them.

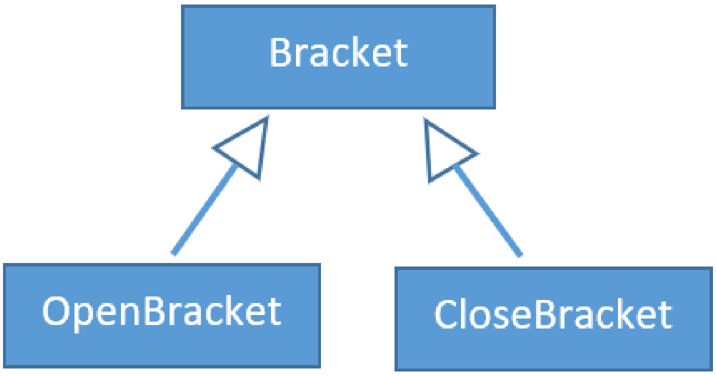
Numbers

I need a class for numbers, 'Num', which will store their value as an attribute.

The value attribute needs to be converted to a floating point as it is a string when extracted from the expression. I could convert it to an integer but that wouldn't allow decimals. I could convert it to an integer or a floating point depending on whether it will lose accuracy or not, but they have the same functionality (you can add, subtract, multiply… them) and when I add settings, I will round the answer anyway.

Brackets

I need a class to identify brackets – open brackets and close brackets. To do this, I will have a parent 'Bracket' class and 2 child classes – 'OpenBracket' and 'CloseBracket' which inherit from it. The inheritance is for cleanliness so if I want to check whether it is a bracket (and not care which), I only have to check once.



Operators

I need a class to identify operators and this will need to store the precedence, associativity and function to execute the operation. I will instantiate this class in the valid tokens dictionary and even though the same instance may be referenced multiple times if the same type of operator is in an expression more than once, it doesn't matter as they will both act exactly the same – I won't be storing the operands in the class but just calling a method to execute it, passing the operands as parameters.

Attribute Name	Datatype	Description
name	String	The name of the operator so I can identify it when debugging (may remove it at the end)
func	Identifier of function	The function to execute the operation
precedence	Integer	The precedence of the operator type as shown in the operator precedence table (lower number is better precedence and executed first)

is_left_associative	Boolean	Whether or not the operator is left associative (alternative is right associative)
is_unary	Boolean	Whether or not the operator is unary – has 1 operand

Table 3: Operator Attributes

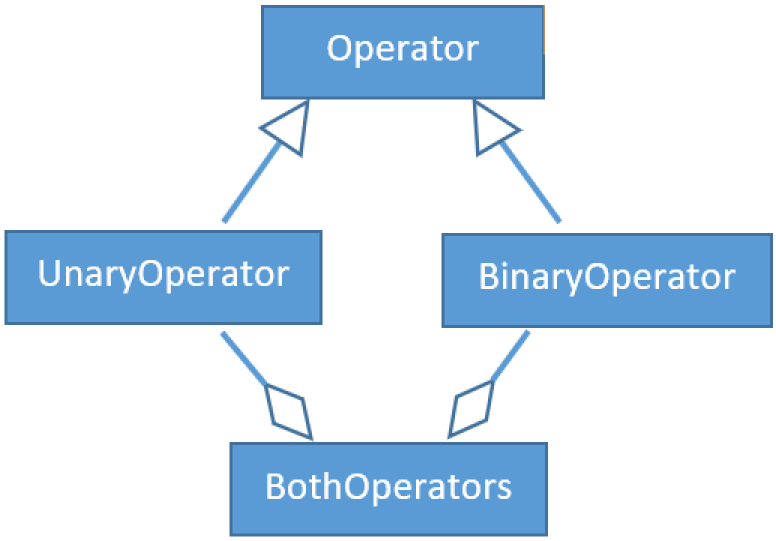
I will also make a method 'execute' which will take an array of operands, call the function to execute the operation on these and return the answer.

The code for this class is mainly storing these attributes and very language-specific so I won't show it as pseudocode.

I need a distinct class for each type of token I may need to check for - unary and binary operators need to have different classes. This means the class described above will be the base class 'Operator' and I will have 2 child classes 'UnaryOperator' and 'BinaryOperator' that inherit from it.

Some symbols are 'overloaded' as explained here[3] such as '+' and '-' which means I need to identify which operator they represent based on the context. I will write an algorithm for this later but at this stage, I need to identify it as requiring this algorithm by making it an instance of 'BothOperators' meaning it could be either and I need to work out which.

This 'BothOperators' class will use aggregation rather than inheritance to associate with 'BinaryOperator' and 'UnaryOperator'. It will have a 'unary' attribute to store an instance of 'UnaryOperator' and a 'binary' attribute to store an instance of 'BinaryOperator'. 'BinaryOperator' and 'UnaryOperator' both exist outside 'BothOperators', so it is aggregation not composition.



To store tokens during conversion and execution of expressions

In 'convert', I will use the shunting yard algorithm which uses both stacks and queues, and in 'execute', I will use a new stack and the queue of tokens from 'convert' to execute the expression in order.

The following classes encapsulate the operations of stacks and queues into a single place and make it easy to interact with.

Stack Methods

Method name	Description
push(item)	Adds 'item' to the top of the stack
pop()	Removes from the stack and returns the item at the top of the stack unless the stack is empty
peek()	Returns the item at the top of the stack unless the stack is empty

Table 4: Stack Methods

[3] http://www.cs.man.ac.uk/~pjj/cs211/ho/node2.html under the 'Overloading' subheading

Queue Methods

Method Name	Description
enqueue(item)	Adds 'item' to the back of the queue
dequeue()	Removes from the queue and returns the item at the front of the queue unless the queue is empty
peek()	Returns the item at the front of the queue unless the queue is empty

Table 5: Queue Methods

'tokenise'

Splits the expression into tokens and makes them an instance of the relevant class. This is surprisingly difficult so I will use regular expressions as they match strings to patterns.

Regex

I obviously need to identify numbers and brackets as tokens, but it is less obvious what to do with the rest. I could hard-code each operation into the regex and identify it from there, but this would not allow users to add their own operations very easily.

As my operators are all a single character, I can identify anything other than a number and bracket as 'other' and decide whether it is valid or not and what operator it is when I identify it.

Despite it not affecting the expression, I will also identify whitespace (any character that you can't see) just so it is separated from other characters – I will ignore it later.

In summary, the different regex patterns I will identify are:

- **Whitespace**: Any character you can't see
- **Number**: Integers or decimals
- **Bracket**: An open or close bracket
- **Other**: Anything that hasn't matched the above

Identifying Tokens

Later in the calculator, I need to know what type of token each token is. To do this, I will make them an instance of a class, so I can store the code that executes them and their associativity and precedence in it.

Below are the operators I will need in the dictionary to start with (the minimum operators to test on):

Symbol	Binary/Unary	Name	Precedence	Associativity
-	Binary	Subtraction	4	LEFT
-	Unary	Negative	2	RIGHT
!	Unary	Factorial	2	LEFT
^	Binary	Exponentiation	1	RIGHT

Table 6: Starting Operator Attributes

To identify the tokens, I need to make it an instance of the relevant class given the name and value (string that was matched):

1) If it is a number, make it an instance of my number class.
2) If it is a bracket, make it an instance of either:
   a) my open bracket class if it is an open bracket
   b) my close bracket class if it is a close bracket
3) If it is in my 'valid_tokens' dictionary, add this to the list (if it could be unary or binary, decide which and add that).
4) If it is invalid, raise an error (type 1)

Name	Datatype	Role	Description and Purpose

name	String	Fixed value	The name of the pattern the token matched to in the regex, so I know what to identify it as
value	String	Fixed value	The part of the expression that was matched by the regex, so I know what to identify it as
prev_token	Custom object	Fixed value	The instance of the class the last token was identified as so I know what type of token the last token was. The last token can affect what token the current token is
valid_tokens	Dictionary	Fixed value	The 'valid_tokens' dictionary that stores all the allowed tokens so I can check to see whether this token is valid by checking if it's in there
token	Custom object	Fixed value	The instance of the class the current token is identified as - what I am trying to find and return
operator	Custom object	Fixed value	The instance of the operator class that the current token is - saved as a variable so I don't have to keep accessing the 'valid_tokens' dictionary

Table 7: Variable Roles for 'identify'

```
FUNCTION identify(name, value, prev_token, valid_tokens)
    IF name = "number"      // 1
        token ← new number
    ELSEIF value = "("      // 2a
        token ← new open bracket
    ELSEIF value = ")"      // 2b
        token ← new close bracket
    ELSEIF value IN valid_tokens      // 3
        operator ← valid_tokens[value]
        IF operator could be unary or binary
            IF should_be_unary(prev_token)
                token ← operator.unary
            ELSE
                token ← operator.binary
            ENDIF
        ELSE
            token ← operator
        ENDIF
    ELSE
        RAISE error invalid token
    ENDIF
    RETURN token
ENDFUNCTION
```

'should_be_unary' is a helper function defined below.

An operator should be treated as unary if:

1) it is the first token in the expression
2) it is preceded by an open bracket
3) it is preceded by an operator that:
    a) is binary
    b) or is unary and right-to-left associative

Name	Datatype	Role	Description and Purpose
prev_token	Custom object	Fixed value	The instance of the class the previous token was identified as which may determine what type of token the current token is identified as

Table 8: Variable Roles for 'should_be_unary'

```
FUNCTION should_be_unary(prev_token)
```

```
        IF prev_token is null      // 1
                RETURN true
        ELSEIF prev_token is an open bracket     // 2
                RETURN true
        ELSEIF prev_token is an operator          // 3
                IF prev_token.is_unary = false    // 3a
                        RETURN true
                ELSEIF prev_token.is_left_associative = false      // 3b
                        RETURN true
                ELSE
                        RETURN false
                ENDIF
        ELSE
                RETURN false
        ENDIF
ENDFUNCTION
```

Getting tokens from regex and identifying them

The regex only defines the pattern to search for. I now need to match this to the expression, identify the tokens and add them to a list.

The way regex works (at least how the python 're' library works) is by matching the start of the string to one of the patterns and when it stops following the pattern, it returns the string up to that point that followed the pattern and which pattern it followed. I don't want it to only find 1 match in the string but find all tokens in it – it needs to repeat with the rest of the string. The following pseudocode keeps track of a current position through the string and matches repeatedly until the end of the string.

Due to the 'other' pattern in the regex, every part of the expression will match to something, so I don't need to worry about not having matches.

Ignore whitespace at this stage as it is not a token.

Then pass the name ('key' from find_matched_key), value (string that matched it – 'match[key]') and the previous token (if there is one – if the tokens list isn't empty) to identify to make it an instance of the relevant class and then add this to the list.

Name	Datatype	Role	Description and Purpose
expr	String	Fixed value	The expression the calculator was called with to split into identified tokens
tokens	List of custom objects	Container	Accumulates the tokens in the expression by storing instances of my custom classes to return once all tokens are added
pos	Integer	Stepper	Holds the position through the expression of the start of the current token so I know where the current token starts
match	Dictionary	Most recent holder	Contains all patterns in the regex and the string that matched them, so I know which string matched which pattern. Obtained from the regex
key	String	Most recent holder	The key of the dictionary that was matched which is also the name of the regex pattern that was matched
prev_token	Custom object	Most recent holder	The instance of the class the previous token was identified as to which may help determine what the current token is identified as

Table 9: Variable Roles for 'tokenise'

```
FUNCTION tokenise(expr)
        tokens ← empty list
```

```
    pos ← 1
    WHILE pos <= LEN(expr)
        match ← match 'expr' to 'regex' starting 'pos' characters into it
        pos ← end position of match
        match ← convert to dictionary
        key ← find_matched_key(match)
        IF key ≠ "whitespace"
            IF LEN(tokens) = 0
                prev_token ← null
            ELSE
                prev_token ← tokens[-1]        // last token to be added to tokens
            ENDIF
            token ← identify(key, match[key], prev_token)
            add token to tokens
        ENDIF
    ENDWHILE
    RETURN tokens
ENDFUNCTION
```

find_matched_key is a helper function that finds which name from the regex was matched. This is needed because when converting to a dictionary, every name from the regex is added as a key with the string it matched as a value. However, it can only match 1 of these so all but 1 value in the dictionary is 'null'. This means even though it seems as if the return statement could be executed more than once, it will be executed exactly once – no more than once because of the nature of regex (it can only match to one thing) and not 0 times because my last regex category matched anything else.

Name	Datatype	Role	Description and Purpose
match	Dictionary	Fixed value	The patterns from the regex and the strings that matched to them so I can find which pattern was matched
key	String	Most recent holder	The key in the dictionary which is the pattern in the regex to see if that pattern was matched to something

Table 10: Variable Roles for 'find_matched_key'

```
FUNCTION find_matched_key(match)
    FOR key IN match
        IF match[key] ≠ null
            RETURN key
        ENDIF
    ENDFOR
ENDFUNCTION
```

'convert'

Before I can execute this expression, the calculator needs to know the order in which to execute the operations. This is where precedence and associativity are used to put the expression into postfix (AKA Reverse Polish) notation which means they can be executed in the correct order without needing any brackets.

To do this, I will use the shunting yard algorithm to get a list of tokens in postfix notation.

Shunting Yard Algorithm

Name	Datatype	Role	Description and Purpose
tokens	List	Fixed value	Stores the tokens in the order they were in the expression to be converted into RPN
stack	Stack	Container	Stores all operators before they are added to the queue which helps to determine the order of operations

queue	Queue	Container	The tokens in RPN that needs to be returned once all tokens have been added
token	Custom object	Most recent holder	The current token being processed from tokens so I can decide what to do with it

Table 11: Variable Roles for 'convert'

```
FUNCTION convert(tokens)
        stack ← empty stack
        queue ← empty queue
        FOR token IN tokens
                IF token is a number
                        add token to queue
                ELSEIF token is an operator
                        WHILE the operator at the top of the stack should be executed before the current token
                                move the token from the top of the stack to the queue
                        ENDWHILE
                        add token to stack
                ELSEIF token is an open bracket
                        add token to stack
                ELSE
                        WHILE operator at top of stack is not an open bracket
                                move the token at the top of the stack to the queue
                        ENDWHILE
                        remove the open bracket from the top of the stack
                ENDIF
        ENDFOR
        WHILE LEN(stack) > 0
                move token at the top of the stack to the queue
        ENDWHILE
        RETURN queue
ENDFUNCTION
```

Should be executed first

If the current token is an operator, I need to determine whether or not the token at the top of the stack needs to be executed before the current operator:

The token at the top of the stack should be executed first if both of the following:

1) The token at the top of the stack is an operator (the stack could be empty, or the token could be an open bracket)
2) Either of the following:
    a) The operator at the top of the stack has better precedence than the current operator
    b) Both of the following:
        i) They have equal precedences
        ii) The operator at the top of the stack is left-to-right associative

'execute'

Executes the expression which is in reverse polish notation to get the answer.

Whenever I encounter a number, I add it to the stack.

Whenever I encounter an operator, I remove the number of operands it needs from the stack. Then I execute the operation on the operands to get a numerical answer and add this to the stack.

Once done for everything in the expression, there should be a single item in the stack which is the final answer. If this isn't the case, error because the user has made a mistake in the expression.

Name	Datatype	Role	Description and Purpose
tokens	Queue	Fixed value	The queue of tokens in RPN from 'convert'. Needed to know the order to execute them
stack	Stack	Container	Stores the operands before they are executed by an operator
token	Custom object	Most recent holder	The instance of the class the token was identified as so I can decide what to do with it
operands	List	Container	Stores the operands needed for an operation before it is executed with them

Table 12: Variable Roles for 'execute'

```
FUNCTION execute(tokens)
        stack ← empty stack
        FOR token IN tokens
                IF token is a number
                        add token to stack
                ELSE    // must be an operator
                        operands ← []
                        move operand on top of stack to operands
                        IF operator is binary
                                Move another operand from the top of the stack to operands
                        ENDIF
                        execute token with operands and add to stack
                ENDIF
        ENDFOR
        IF LEN(stack) ≠ 1
                ERROR as there is the wrong number of operands/operators
        ELSE
                RETURN item on stack
        ENDIF
ENDFUNCTION
```

'calculate'

This is not a helper function but the whole function that is executed when someone calls the calculator with an expression, wanting an answer.

It links 'tokenise', 'convert' and 'execute' together so when the calculator is called with an expression, it passes it to each of these in turn, checking for errors as it goes.

I will put them in an array and iterate over it because I do the same thing with each of them and I can easily add more.

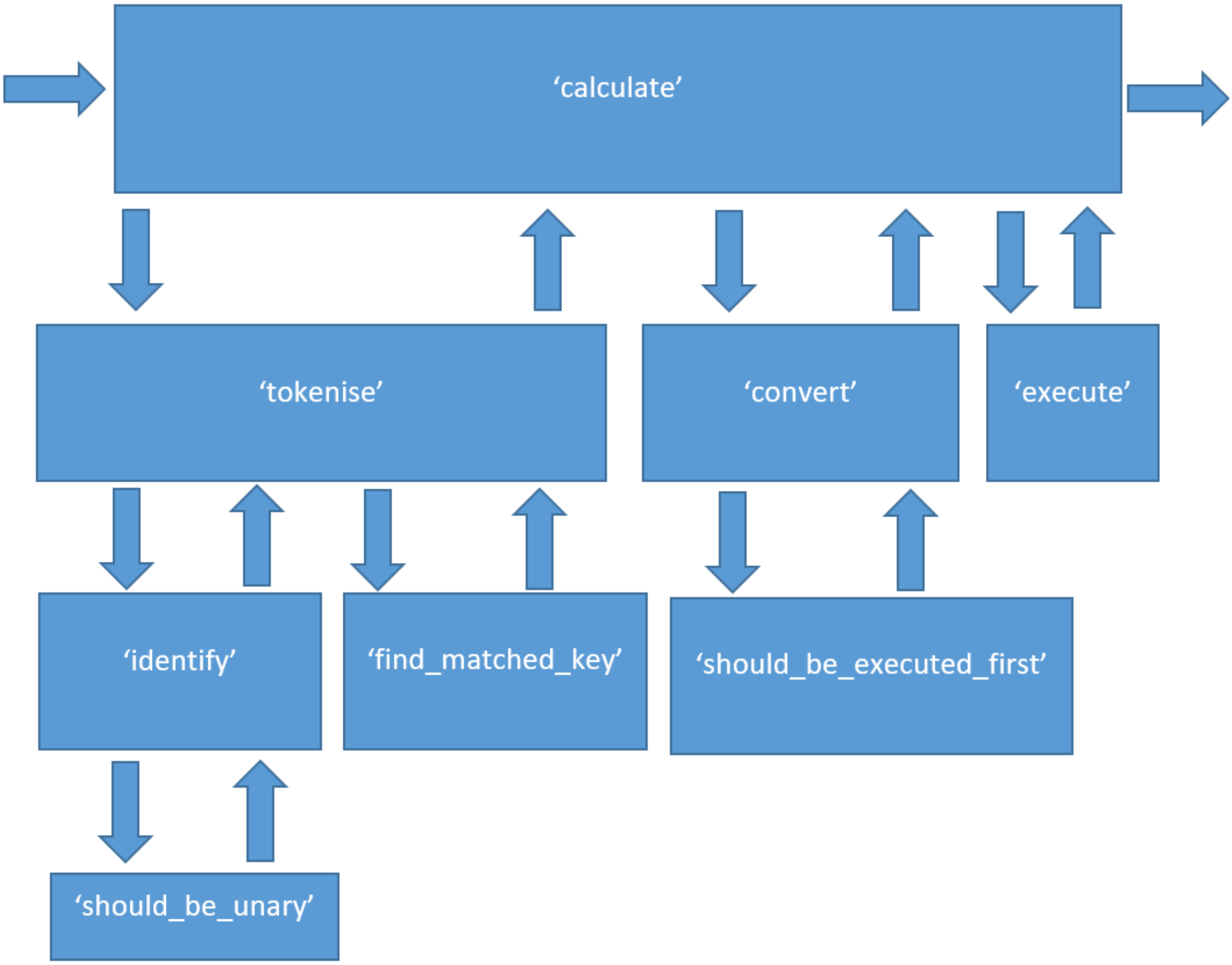
The variable 'expr' changes as it is fed through each function in turn. It is passed to each function in turn and it is replaced with whatever they return to be passed to the next function. At the end, it will be the answer

Name	Datatype	Role	Description and Purpose
expr	Starts as a string but varies	Most recent holder	The input and output to each function in 'funcs' which process it, getting it closer to the answer
funcs	List	Fixed value	The functions to process the expression
func	Function	Most recent holder	The function currently being executed

Table 13: Variable Roles for 'calculate'

Calculator

```
FUNCTION calculate(expr)
        funcs ← [tokenise, convert, execute]
        FOR func IN funcs
                expr ← func(expr)
        ENDFOR
        RETURN expr
ENDFUNCTION
```



2) Interface (Memory)

The interface will have its own class, so it can store the memory as an attribute and provide methods which perform the required functionality.

Method Name	Description
constructor()	Initialise the memory as empty
calculate(expr)	Calculate the answer to an expression by calling the main calculator (with debug off) but saving the expression and answer in memory
len_memory()	Return the number of items in memory
memory_item(num_calculations_ago)	Return a specific item in memory by its number which is how recently it was saved (1 being the most recent and ascending from there)
recent_memory(num_to_retrieve)	Return a specified number of the most recent items in memory. Can be null, meaning all items in memory
clear_memory()	Clear the memory

Table 14: Interface Methods

The datatype to store the actual memory will be an array as it is an ordered collection of items. It cannot be a stack or queue as despite only needing to add items to the front, I could need to retrieve any of the items in it, and I don't want to remove them when I do retrieve them. I want to store both the answer and the original expression in memory, so it will be an array of 2-value arrays with the first being the expression and the second being the answer.

I could also provide a method to parse an expression, searching for 'ans' to replace it with the most recent answer, however different interfaces will work differently so I will not include it. A command-line interface may do it this way, but a graphical user interface may have a button which shows all memory items and when the user clicks on one, it will insert it into the expression.

The code will be simple, mainly checking for invalid parameters and quite language specific so I will not provide pseudocode as the explanations above do most of the work.

Instructions

I need to provide instructions for users of the calculator despite not knowing what part of the calculator they will be using – they may be using my user interface, or my memory, or someone else's user interface, etc. Because of this, each file should assume the user is using that file directly (rather than a file that imports it) and provide instructions relevant for that file only. Any files building on it can import these instructions and extend them to include information specifically about that file. This will only be instructions for users rather than programmers as docstrings are for the programmers.

Testing Interface

In order to test this section, I need a command-line interface. It will search for keywords in the expression which will correspond to a method (or attribute in the instructions case):

Keyword	Method call	Description
memory	recent_memory()	Display all recent memory items – expressions and answers
Mx	memory_item(x)	Replace 'Mx' with the xth memory answer
clear	clear_memory()	Clear the calculator's memory
instructions	instructions	Display the calculator's instructions
ans	memory_item()	Replace 'ans' with the most recent memory answer

Table 15: Testing Interface Keywords

This can also act as a command-line user interface for users if they want to use the calculator without a GUI.

3) Graphical User Interface

GUI Layout

The main view of the GUI will have buttons along the top allowing you to:

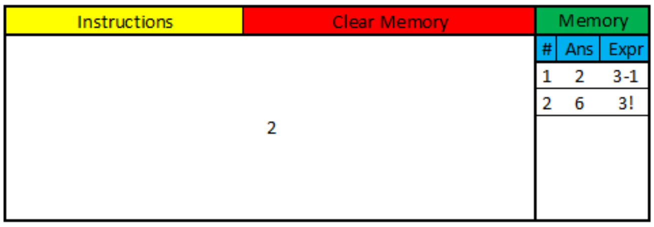
- View the instructions on a separate screen
- Clear the memory

On the right-hand side, there will be a list of all recent calculations – the answers and expressions that created them which you will be able to click on to insert that item into the expression. It will insert 'Mx' into the current point in the expression where x is the number of the entry clicked on. The user will also be able to type 'ans' or 'Mx' into the expression to have the same effect ('ans' is the same as 'M1').

Under the buttons and to the left of the memory, there will be an open space to type in expressions which will display the text typed. When ENTER is pressed, it will change to the answer and insert that calculation into memory on the right-hand side.

If an error occurs when executing the expression or parsing it for ‘ans’ or ‘Mx’, it will not add anything to memory and display the error message at the top of the expression-entry box.

Proposed design:



In instructions mode, there will be the instructions and a back button, allowing you to go back to the normal mode.

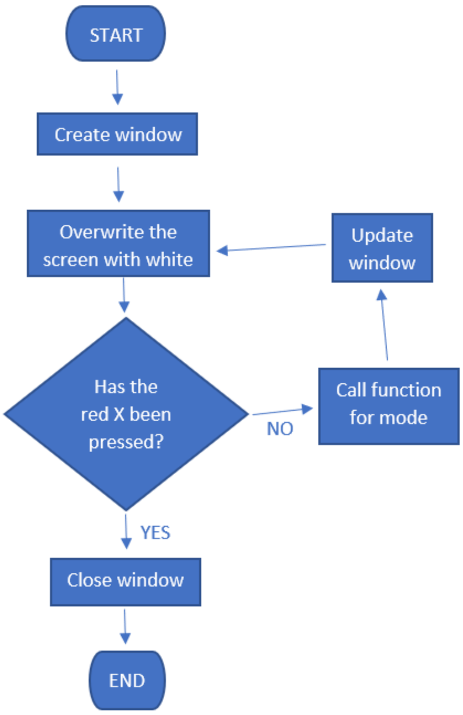
Code Design

I will model the GUI in a class so I can store constants as attributes that all methods have access to. The constructor will initialise constants as well as variables for the calculator and I will create a ‘run’ method which will create the window and start the main loop. Each mode in the calculator will have a method prefixed with ‘mode_’ which is called every tick from inside the main loop of ‘run’ to do what needs to be done when the calculator is in that mode. Things that need to be done regardless of mode are done in the ‘run’ method, so I don’t have to repeat code in each of the mode methods such as:

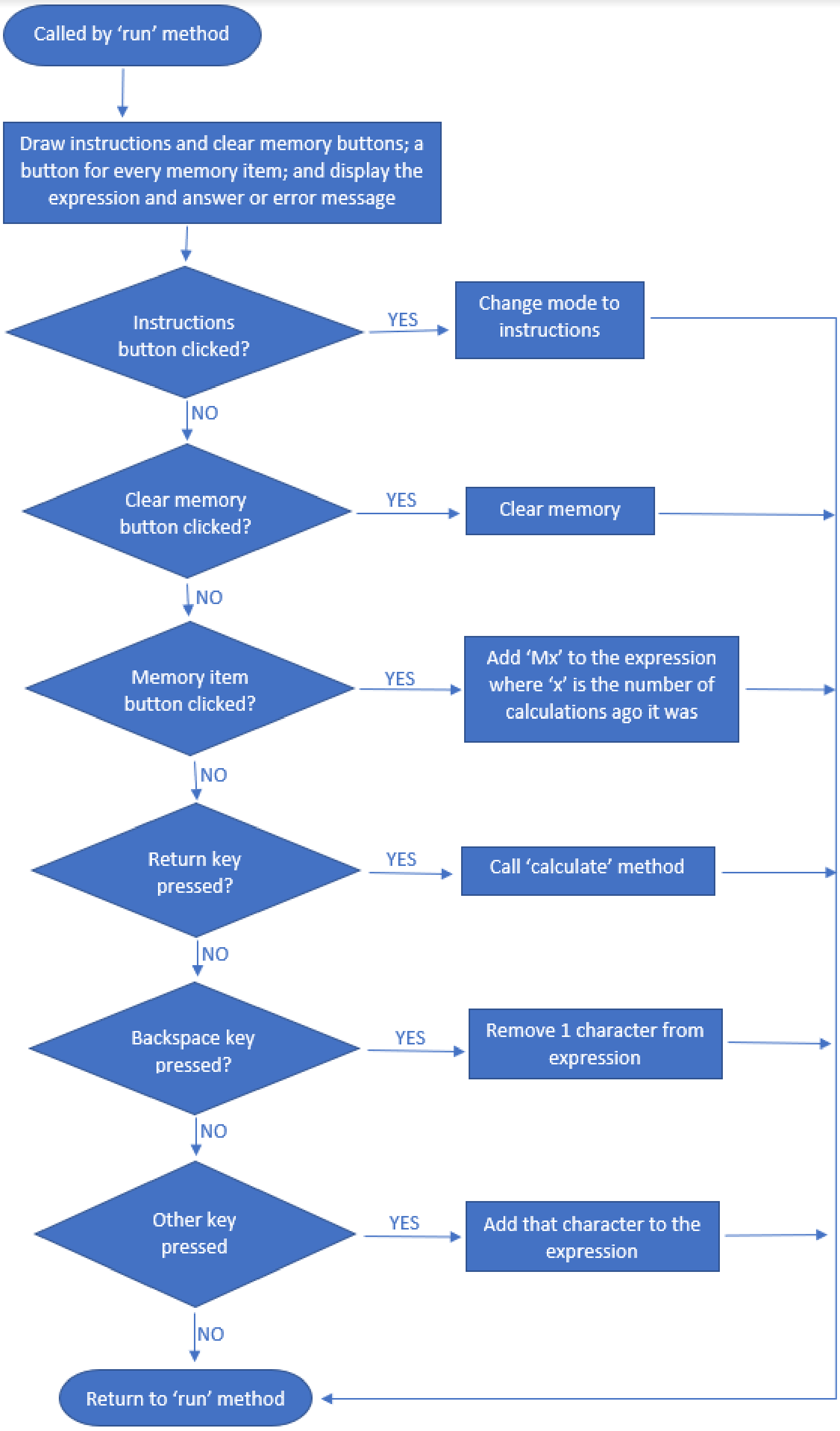
- Overwriting the screen with the background colour so the last frame doesn’t show
- Checking whether the window has been told to close by the operating system (mostly when the red x in the top-right corner has been pressed) and closing the window if so
- Updating the screen

Below are flowcharts for each method in the class explaining what they will do.

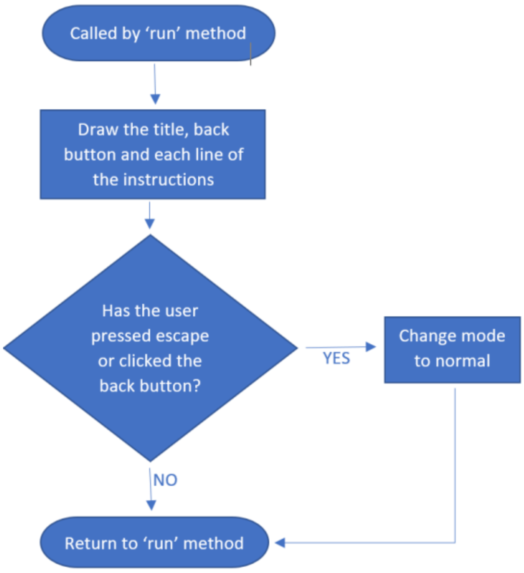
Run Method



Normal Mode Method



Instructions Mode Method



Calculate Method

When the user submits an expression for calculation, this user interface must do lots of processing as well as calling the main calculator to work out the answer which is why I have made it a separate method. As I have implemented referencing memory by typing 'ans' or 'Mx' or clicking on a memory item (which inserts 'Mx' into the expression so acts the same as them typing it), I need to replace these with the actual answer referenced before calling the calculator. To do this, I will do a similar thing to what I did in the command-line interface from stage 2:

1) Lower the string to ignore case
2) Replace 'ans' with 'm1'
3) For each 'm' in the expression:
    a) Find the string after it but before the next non-digit character
    b) If this is valid (a number between 1 and the number of items in memory), replace the reference with the actual answer

If the reference is not valid, I won't error at this stage as it could be part of a word which means something to the main calculator. I will let it stay in the expression and if it is indeed wrong, the main calculator will error, and I will show the error message to the user anyway.

Once I have done the processing, I will pass the expression to the interface which will in turn pass it to the main calculator and get the answer as a result. I will catch any type 1 errors and edit the variables 'expr', 'ans' and 'error' accordingly.

Text and Buttons

To make it faster to draw buttons and display text, I will create my own objects that represent text and buttons that can be created before they are drawn. This enables me to do lots of the processing just once at the start of the program so I only have to draw them while the window is running which will save lots of time and processing power. I don't even have to draw all of them at the same time – I can create all text and buttons for all modes and then only draw them in the modes they need to be in.

This works well for most things, but the memory buttons, expression, answer and error message will change throughout the running of the program. It is not much of a problem to add a method to the text and button objects to change the text, but I need to call these at the right time – if I call them every tick, it will be almost pointless creating them in advance.

For this, I will create a method in the GUI that will update the text/buttons that are out of date. It will take Boolean variables which represent whether or not each text/button needs updating and it will update the text on them. This can be called whenever something needs updating, specifying what needs to be updated so that not everything is updated when 1 thing needs updating.

When I update the memory buttons, I will get the memory items straight from the interface but the expression, error message and answer will be attributes of the GUI so need to be changed themselves before the text objects are updated.

The following table shows when the attributes need to be changed and when the update text and buttons method needs to be called. Some of the changes to the attributes are already stated in the flowcharts above but I will repeat them here to be exhaustive. Everything below applies to normal mode only:

Time	What needs to be updated
When the clear memory button is pressed	Change the error message and answer attributes to empty strings. Update the text objects for the error message and answer and update the memory button objects.
When a memory button is pressed	Add 'Mx' to the expression attribute. Update the expression text object
When a keyboard character is pressed (apart from escape, backspace and return)	Add the character to the expression attribute. Update the expression text object.
When the escape keyboard character is pressed	Change the expression attribute to an empty string. Update the expression text object.
When the backspace keyboard character is pressed	Remove the last character from the expression attribute. Update the expression text object.
When the return keyboard character is pressed with a valid expression	Make the error and expression attributes empty strings and the answer attribute the answer from the main calculator. Update the expression, answer and error message text objects and the memory button objects
When the return keyboard character is pressed with an invalid expression	Make the answer attribute an empty string and the error message attribute the error message from the main calculator. Update the answer and error message text objects

Table 16: When to update the text and button objects

4) Constants

A constant is some string of letters that represents a number, e.g.: ‘e’ (Euler’s number = 2.718…) and ‘π’ (3.14159…). However, there isn’t a ‘π’ character on the keyboard, so I will have to represent it with ‘pi’.

Constants can be added to the ‘valid_tokens’ dictionary with the string they will be referenced by as the key and a ‘Num’ object as the value. There is no need to create a constant class as it is just a number represented by letters.

To test this, I will use ‘pi’ for a multi-letter constant and ‘e’ to check the standard from below won’t interfere with it.

5) Standard Form

Until now, I have deliberately ignored standard form including testing very big or small numbers as they will be written in standard form. One problem is that by default, Python writes standard form with an ‘e’, but I plan to include ‘e’ as Euler’s number. This means I need an alternative notation for standard form in my calculator and not only do I need to parse this notation when the user inputs it, but when a calculation is executed, I need to convert it from the ‘e’ form to my notation before outputting it to the user.

In my calculator, the equivalent to ‘e’ in python and ‘x10^’ in maths, meaning ‘times 10 to the…’ will be ‘~’. This tilde character is not used for much else in maths so probably won’t interfere with other calculations.

I need to convert ‘~’ back into ‘e’ so that python can understand it, but I need to be careful not to mix Euler’s number ‘e’ up with it. To do this, I can replace ‘~’ with ‘e’ in ‘identify’ for inputs that have been matched to the ‘number’ pattern. This will not replace interfere with Euler’s number as this will take the value in ‘valid_tokens’.

To convert back, at the end of ‘execute’, I can replace ‘e’ with ‘~’ again so the output to the user is consistent and if they use this answer in a further operation, the calculator can understand it.

6) Functions

Datatypes

Functions need 2 layers of classes as not only do I need an instance of a function class for each *type* of function (e.g.: sine, cosine, etc.), but I also need an individual instance of each of these that occur in the expression – I may have 2 sine’s in 1 expression and these need to be different instances as I will be storing the operands in the class and if they were the same instance, the operands would override each other.

This principal is known as a factory class – a class that’s purpose is to create another class. I will instantiate the factory class once for each *type* of function (sine, cosine, etc.) so I can store it in the ‘valid_tokens’ dictionary, no matter if it appears in the expression. This class will store code to execute the function as an attribute and implement a method to create an instance of a new class, of which there will be an instance for each function in the expression. This second class will be passed the code to execute it but will also be able to store the operands.

The user doesn’t have to put single numbers as operands for functions – they could input expressions instead. This means I need to recursively call the calculator with each operand before executing the function on the operands. I will do this in the ‘execute’ method of the ‘FunctionInstance’ class before executing the function on its simplified operands.

I will call the factory class ‘FunctionType’ and it will have the following attributes and method:

Attribute Name	Datatype	Description
‘name’	String	The name of the function so I can identify it when debugging (may remove it at the end as non-essential)
‘func’	Identifier of function	The function to execute the operation

Table 17: ‘FunctionType’ Attributes

Method Name	Description	Parameters	Returns
Constructor	Instantiates the class	'name' – the name of the type of function 'func' – the function to execute the operation	
'create'	Creates a 'FunctionInstance' object for each instance of the function in the expression	'calc' – the calculate function from the main calculator	A 'FunctionInstance' object

Table 18: 'FunctionType' Methods

I will call the actual function class 'FunctionInstance' and it will have the following attributes and methods:

Attribute Name	Datatype	Description
'name'	String	The name of the function so I can identify it when debugging (may remove it at the end as non-essential)
'func'	Function	The function to execute the operation
'calc'	Function	The calculate function from the main calculator
'operands'	Array	The operands of the function

Table 19: 'FunctionInstance' Attributes

Method Name	Description	Parameters	Returns
Constructor	Instantiates the class	'name' – the name of the type of function 'func' – the function to execute the operation 'calc' – the calculate function from the main calculator	
'add_operation'	Executes the operand using the calculate function from the main calculator and adds it to 'operands'	'operand' – the operand to add	
'execute'	Recursively execute all operands using the calculator and then return the answer when the function is executed with these operands		The answer to the function when executed on its operands

Table 20: 'FunctionInstance' Methods

Regex

The operands will be separated by commas, so I need to identify these separately in the regex. To do this, just add a new pattern called 'comma' that matches exactly 1 ',' character.

Identify

If there is a comma in the expression outside a function, it is invalid so needs to give an error message and when a token is a 'FunctionType', I need to call the 'create' method and return the 'FunctionInstance'.

Tokenise

In 'tokenise', I need a few new variables for controlling functions:

Name	Datatype	Role	Description and Purpose
'in_func'	Boolean	Most recent holder	Whether or not 'pos' is in a function so I know whether to add operands or not
'bracket_depth'	Integer	Most recent holder	How many bracket pairs the current position is inside of, allowing me to check brackets match properly

'operand_start_pos'	Integer	Most recent holder	The position through the expression that the current operand started so I can get the operand from the expression once at the end

Table 21: New Variables in 'tokenise'

When 'tokenise' receives a token from 'identify' that is a 'FunctionInstance', it should:

- Set 'in_func' to 'true' – so I know I'm in a function
- Initialise 'bracket_depth' to 0 – as it is relative to the function
- Initialise 'operand_start_pos' to 'pos' – so I know where the first operand starts

When 'in_func' is 'false', I do the previous tokenise algorithm, however when it is 'true', I need a new algorithm.

Any characters apart from brackets and commas, I can ignore as when I get to the end of the operand, I can get them in 1 go using 'operand_start_pos'.

- If it is an open bracket, increment 'bracket_depth'
- If it is a close bracket, decrement 'bracket_depth'
- If it is a comma, it is the end of the operand with another operand coming next so:
  - Identify and add the operand to the function
  - Update 'operand_start_pos' to the current position
- If 'bracket_depth' is now '0', it is the end of the operand and whole function so:
  - Change 'in_func' to 'false'
  - Identify and add the operand to the function
  - Execute the function with its operands and replace it in the tokens list

To identify and add the operand to the function, I have created a new helper function 'identify_operand'. When I get the operands using 'operand_start_pos' and the current position, I include the commas and brackets either side so I can check the syntax is correct. This means, the first character of each operand must either be an open bracket or comma, otherwise, provide an error message. Similarly, the last character of each operand must either be a comma or a close bracket, otherwise, provide an error message. Then I can add it to the function object if there hasn't been an error and once I've removed these characters.

Testing Functions

In order to test the calculator handles functions correctly, I need functions with 1, 2 and 3 parameters to test when operands start and end in brackets, only start or only end in brackets, and start and end in commas:

Name	Parameters	Description
'sin'	'theta' – an angle in radians	The sine circular function which finds the ratio between the opposite and hypotenuse sides in a right-angled triangle with respect to angle theta
'root'	'root' - root	Finds the rootth root of x
	'x' - answer	
'sum'	'x' - number 1	Finds the sum of the 3 numbers x, y and z
	'y' - number 2	
	'z' - number 3	

Table 22: Testing Functions

7) Operations

Most operations I need to add have already been written so there is no point writing my own. However, I do need to check for any invalid operations, e.g.: division by 0, and present them as an error message to the user. To do this, I need to find the valid domains and ranges (co-domains) for each of these so I know what to disallow.

Operation Invalid Domains

Operation	Expression	Invalid domains
Addition	x+y	
Subtraction	x-y	
Multiplication	x*y	
Division	x/y	y=0 – division by 0 is undefined
Floor division	x\y	y=0 – division by 0 is undefined
Mod	x%y	y=0 – division by 0 is undefined
Exponentiation	x^y	x=0 AND y=0 – 0 to the power of 0 is undefined
Root	x¬y	x<=0 – cannot find the negative or 0^{th} root of a number x is not a whole number – only deals with discrete roots of a number
Combination	nCr	n<0 – cannot have a negative number of places r<0 or n<r – r must be between 0 and n r or n are not whole numbers – only deals with discrete values
Permutations	nPr	n<0 – cannot have a negative number of places r<0 or n<r – r must be between 0 and n r or n are not whole numbers – only deals with discrete values
Positive	+x	
Negative	-x	
Factorial	x!	x is not a whole number – only deals with integers x<0 – only deals with positive values
Natural log	ln(x)	x<=0 – solutions to exponents are always >0
Logarithm	log(x, base)	x<=0 – solutions to exponents are always >0 base<=0 – bases must be positive
Absolute value	abs(x)	
Lowest common multiple	lcm(x, y)	x or y are not whole numbers – only deals with integers x<1 or y<1 – only deals with positive numbers
Highest common factor	hcf(x, y)	x or y are not whole numbers – only deals with integers x<1 or y<1 – only deals with positive numbers
Quadratic equation solver	quad(a, b, c)	b^2-4ac<0 as this leads to a negative square root which leads to imaginary numbers which I am not including
Random Integer	rand(low, high)	low>high low or high are not whole numbers – only deals with integers
Sine	sin(x)	
Cosine	cos(x)	
Tangent	tan(x)	cos(x)=0 therefore x%pi=0.5*pi – division by 0 is undefined
Inverse sine	arsin(x)	x<-1 or x>1 – the co-domain of sin(x) is only between -1 and 1
Inverse cosine	arcos(x)	x<-1 or x>1 – the co-domain of cos(x) is only between -1 and 1
Inverse tangent	artan(x)	
Hyperbolic sine	sinh(x)	
Hyperbolic cosine	cosh(x)	
Hyperbolic tangent	tanh(x)	
Inverse hyperbolic sine	arsinh(x)	
Inverse hyperbolic cosine	arcosh(x)	x<1 – the sum of e^x and e^{-x} has minimum value 2 so halved, this is 1
Inverse hyperbolic tangent	artanh(x)	x<=-1 or x>=1 – $(e^{2x}-1)/(e^{2x}+1) \rightarrow -1$ as $x \rightarrow -\infty$ and 1 as $x \rightarrow +\infty$

Table 23: Operation Invalid Domains

Operator Precedence and Associativity

Name	Symbol	Number of Operands	Precedence	Associativity

Addition	+	2	4	Left-to-right
Subtraction	-	2	4	Left-to-right
Multiplication	*	2	3	Left-to-right
Division	/	2	3	Left-to-right
Floor division	\	2	3	Left-to-right
Mod	%	2	3	Left-to-right
Exponentiation	^	2	2	Right-to-left
Root	¬	2	2	Right-to-left
Combinations	C	2	0	Left-to-right
Permutations	P	2	0	Left-to-right
Positive	+	1	1	Right-to-left
Negative	-	1	1	Right-to-left
Factorial	!	1	1	Left-to-right

Table 24: Operator Precedence and Associativity

Constant Values

Symbol	Value
pi	3.14159265358979323846264338327950288
tau	6.28318530717958647692528676655900576
e	2.71828182845904523536028747135266249
g	9.80665
phi	1.61803398874989484820458683436563811

Table 25: Constant Values

Custom Algorithms

The following algorithms don’t have built-in algorithms to Python and don’t require tables of data so I will write my own algorithms for them:

- ‘op_root’ which is the same as a power of 1 divided by the root number so ‘a¬b’ is the same as ‘b^(1/a)’ where ‘^’ means ‘to the power of’:

```
FUNCTION root(a, b)
        RETURN b ^ (1 / a)
ENDFUNCTION
```

- ‘op_factorial’ which is the product of all numbers between 1 and the argument, inclusive:

```
FUNCTION factorial(x)
        product ← 1
        WHILE x > 1
                product ← product * x
        ENDWHILE
        RETURN product
ENDFUNCTION
```

- ‘op_permutations’ which is ‘n!/(n-r)!’, using the factorial function already defined

```
FUNCTION permutations(n, r)
        RETURN factorial(n) / factorial(n – r)
ENDFUNCTION
```

- ‘op_combinations’ which is ‘n!/(r!*(n-r)!)’, using the factorial function already defined

```
FUNCTION permutations(n, r)
        RETURN factorial(n) / (factorial(r) * factorial(n – r))
ENDFUNCTION
```

- 'func_quad' – quadratics can produce 2 answers, so I need to split this into 2 different functions – 'func_quadp' for the positive square root answer and 'func_quadn' for the negative square root answer. Both use the quadratic formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- 'func_lcm' and 'func_hcf':

These need more complex algorithms so I will split them into different functions as I need to find the prime factors of each number and get it in a format that lets us see how many of each prime factor I have. The LCM has all prime factors in either number, not both (the maximum number of them in either number), whereas the HCF only has the prime factors in both numbers (the minimum number of the prime numbers in both numbers).

The algorithm to split a number into its prime factors:

It works by trying all numbers between 2 and the square root of the number. For each, it checks whether it is divisible by it and if so, it adds it to the factors list. It only increments the number to try if a match wasn't found so the same prime factor can be found more than once. The last factor is always left as it is smaller than i^2 so I need to add it to the factors too at the end (unless x was 1 at the start in which case there are no prime factors so don't).

Name	Datatype	Role	Description and purpose
x	Integer	Most recent holder	The input and remainder after its previous self has been divided by its prime factors. Needed to know what's left to find factors of
factors	List	Container	Gathers the prime factors to return
i	Integer	Stepper	The factor to try for divisibility. Increments if not divisible

Table 26: Variable Roles for 'prime_factors'

```
FUNCTION prime_factors(x)
        factors ← empty list
        i ← 2
        WHILE i^2 <= x
                IF x % I = 0
                        x ← x / i
                        add i to factors
                ELSE
                        i ← i + 1
                ENDIF
        ENDWHILE
        IF x > 1
                add x to factors
        ENDIF
        RETURN factors
ENDFUNCTION
```

To get the LCM and HCF, I need to know how many of each prime factor I have so I turn it into a dictionary:

```
FUNCTION to_dict(factors)
        factors_dict ← empty dictionary
        FOR factor IN factors
                IF factor IN factors_dict
                        factors_dict[factor] ← factors_dict[factor] + 1
                ELSE
                        factors_dict[factor] ← 1
                ENDIF
        ENDFOR
        RETURN factors_dict
ENDFUNCTION
```

After I have found the prime factors of the LCM or HCF of the 2 numbers, I will need to find the actual value of that number which I find by multiplying all the prime factors together. However, as they will be in the dictionary, I multiply by the key of the dictionary (actual prime factor) to the power of the value of the dictionary (the number of times it occurs):

```
FUNCTION product_dict(dictionary)
        product ← 1
        FOR key IN dictionary
                product ← product * (key ^ dictionary[key])
        ENDFOR
        RETURN product
ENDFUNCTION
```

To find the LCM of 2 numbers, I find the prime factors of each number in dictionary form. As the LCM has all prime factors in either number but not in both, I can start with the LCM having the same prime factors as the first number and then for each prime factor in the second number, if it occurs in the second number more than the first number, add that number of them to the LCM's prime factors:

```
FUNCTION lcm(x, y)
        prime_factors_x ← to_dict(prime_factors(x))
        prime_factors_y ← to_dict(prime_factors(y))
        prime_factors_lcm ← prime_factors_x
        FOR factor IN prime_factors_y
                IF factor NOT IN prime_factors_lcm OR prime_factors_y[factor] > prime_factors_lcm[factor]
                        prime_factors_lcm[factor] ← prime_factors_y[factor]
                ENDIF
        ENDFOR
        RETURN product_dict(prime_factors_lcm)
ENDFUNCTION
```

To find the HCF of 2 numbers, I find the prime factors of each number in dictionary form. As the HCF has the prime numbers in both numbers only, I want the minimum number of each prime factor in either of the 2 numbers:

```
FUNCTION hcf(x, y)
        prime_factors_x ← to_dict(prime_factors(x))
        prime_factors_y ← to_dict(prime_factors(y))
        prime_factors_lcm ← empty dictionary
        FOR factor IN prime_factors_x
                IF factor IN prime_factors_y
                        IF prime_factors_x[factor] < prime_factors_y[factor]
                                prime_factors_lcm[factor] ← prime_factors_x[factor]
                        ELSE
                                prime_factors_lcm[factor] ← prime_factors_y[factor]
                        ENDIF
                ENDIF
        ENDFOR
        RETURN product_dict(prime_factors_lcm)
ENDFUNCTION
```

8) Settings

Unfortunately, I have run out of time to do this section so I will just round all answers to 15 decimal places and have to leave out other settings and the ability to change them.

Technical Solution

Throughout, I will be using the Visual Studio Code IDE with the Python and Bracket Pair Colouriser 'Extensions' and git source control with online Gitlab storage. The IDE shows me docstrings, autocompletes code, syntax highlights and much more which helps me work faster and the extensions allow me to debug, run and see code easily within the application. The source control allows me to track versions of my calculator in case I make a mistake and need to revert to an earlier version and Gitlab allows me to push my versions to the internet, so I can access them from anywhere.

Comments and Docstrings

The most important comments/docstrings are docstrings on public functions that the user may actually use. They need to explain all parameters and their datatypes as well as all return values and their types as well as describe what the actual function does (but not *how* it does it).

To document parameters, I will type ':param' followed by the parameter identifier and then the parameter type in brackets. Then I will explain it as clearly as possible. I will do almost the same with return values but starting with ':return'.

Docstrings in private functions and comments don't need to be as clear but still need to explain how the function works.

I may not add docstrings to magic methods such as '__repr__' as they are only for my benefit when debugging. They mostly format strings to display messages allowing me to track the progress of the calculator and identify where any errors occur.

1) Core Functionality

Errors

I have a parent class 'CalcError' (which in turn inherits from the Python base class 'Exception') that any error within the calculator can take. Although having exactly the same functionality as 'Exception', it is not redundant as I will catch only this error (and any errors that inherit from it) so its purpose does differ from 'Exception'.

```
class CalcError(Exception):
    """General errors in the calculator"""
    # inherits all of 'Exception's methods and attributes
    pass
```

The child class 'CalcOperationError' which holds extra information about errors in specific operations, so the user knows which operation it is from. Normally the error would show a traceback so the programmer would know where the error came from but as I am catching this to display to the user as an error message, they won't know where it is from. This is why it asks for 2 more parameters and assembles the error message with a string format before calling 'Exception's constructor with this.

```
class CalcOperationError(CalcError):
    """
    Errors in the calculator due to specific requirements of operations
    eg: factorial is undefined for negatives or decimals

    :param msg (str): The error message - as clear as possible so someone with no mathematical or programming knowledge can understand
    :param op_name (str): The name of the operation that errored
    :param operands (iterable): The operands the operation was called with
    """
    # inherits all of 'CalcError' and therefore 'Exception's method and attributes but overrides the init method
    def __init__(self, msg, op_name, operands):
        # converts all operands to strings and separates them by commas in the message
        super().__init__("While performing {} with {}: {}".format(op_name, ", ".join([str(operand) for operand in operands]), msg))
```

Datatypes

I will often define a '__repr__' or '__str__' class method which is a magic method that returns a string to represent the class. I will use them for debugging as it will show me the message, so I know exactly what's happening.

Numbers

```
class Num(float):
    """Represents a number"""
    # inherits all methods from float but overrides representation method
    def __repr__(self):
        return "Num({})".format(self)
```

Inherits from the python datatype 'float' which allows 'Num' to inherit all its arithmetic operations and the constructor. I just package it up as a 'Num' and override its '__repr__' magic method which allows me to display it as 'Num' which is what will be shown when debugging it.

Because Python is an object orientated language, how it works with arithmetic operations is when there is an arithmetic symbol such as '+', it looks at the objects of the things either side of it and calls their '__add__' magic method. This method actually implements the addition which is why you can 'add' 2 strings – the dunder (double underscore) add method has implemented it as concatenation. For this reason, inheriting all the arithmetic magic methods from 'float' will allow python to know what to do when it has to add 2 'Num' instances together.

Brackets

The parent bracket class which stores only whether or not it is an open bracket. I could have stored this as a string such as 'open' or 'closed' but it uses less memory as a Boolean and as there are only 2 possibilities, it suits this datatype.

```
class Bracket:
    """
    Represents a bracket

    :param is_open (bool): Whether or not the bracket is an open bracket (alternative is a close bracket)
    """

    def __init__(self, is_open):
        self.is_open = is_open

    def __repr__(self):
        return "OpenBracket" if self.is_open else "CloseBracket"
```

These child bracket classes are non-essential, but they simply make the code more readable as it is clearer which bracket I am referring to when instantiating and checking for these.

The inheritance also means I can check whether either are brackets in general.

The built-in 'super' function returns the direct parent class of the class it was called in. I could have written 'Bracket' instead of 'super()' but super is better practice as this will still work if I were to change the name of the parent class or inherit from a different class.

I considered putting the '__repr__' method in both 'OpenBracket' and 'CloseBracket' and not in the parent 'Bracket' but decided against it as this way, the child classes are unnecessary so if an object was created straight from the parent class, it would still be represented properly (although be less readable when being created).

```
class OpenBracket(Bracket):
    """Represents an open bracket"""

    def __init__(self):
        super().__init__(True)

class CloseBracket(Bracket):
    """Represents a close bracket"""

    def __init__(self):
        super().__init__(False)
```

Operators

Simply saves the attributes explained in design to the instance and implements the 'execute' method.

```
class Operator:
    """
    Represents an operator and stores information about it

    :param name (str): The name of the operator
    :param func (identifier): The identifier of the function to execute the operation
    :param precedence (int/float): The precedence of the operation compared to other operations. Lower numbers means executed first
    :param is_left_associative (bool): Whether or not the operator is left (-to-right) associative (right (-to-left) associative otherwise)
    :param is_unary (bool): Whether or not the operator is a unary operator (takes only 1 operand) or otherwise it is binary (takes 2 operands)
    """

    def __init__(self, name, func, precedence, is_left_associative, is_unary):
        self.name = name
        self.func = func
        self.precedence = precedence
        self.is_left_associative = is_left_associative
        self.is_unary = is_unary

    def execute(self, operands):
        """
        Return the answer when the operator is executed with its operands

        :param operands (list): The operands to execute the operator with
        :return answer (int/float): The answer when the operator is executed with the operands
        """

        # the star splits the 'operands' list out into individual parameters
        return self.func(*operands)

    def __repr__(self):
        return "UnaryOperator({})".format(self.name) if self.is_unary else "BinaryOperator({})".format(self.name)
```

Similarly, to brackets, these child classes are non-essential but make the code more readable. They also slightly simplify it as it determines the is_unary attribute and all unary operators have the same precedence so doesn't need this as a parameter either.

```
class BinaryOperator(Operator):
    """
    Represents a binary operator and stores information about it

    :param name (str): The name of the operator
    :param func (identifier): The identifier of the function to execute the operation
    :param precedence (int/float): The precedence of the operation compared to other operations. Lower numbers means executed first
    :param is_left_associative (bool): Whether or not the operator is left (-to-right) associative (alternative is right (-to-left) associative)
    """

    def __init__(self, name, func, precedence, is_left_associative):
        super().__init__(name, func, precedence, is_left_associative, False)

class UnaryOperator(Operator):
    """
    Represents a unary operator and stores information about it

    :param name (str): The name of the operator
    :param func (identifier): The identifier of the function to execute the operation
    :param is_left_associative (bool): Whether or not the operand is on the left side of the operator (alternative is on the right)
    """

    def __init__(self, name, func, is_left_associative):
        super().__init__(name, func, 2, is_left_associative, True)
```

'BothOperators' is simply to encapsulate a symbol that could be unary or binary in a single object. I check that they are instances of the operator class to catch the error here rather than a different time.

```python
class BothOperators:
    """
    Represents a symbol that could represent a binary operator or a unary operator.

    :param unary (UnaryOperator): The unary operator that it could be
    :param binary (BinaryOperator): The binary operator that it could be
    """

    def __init__(self, unary, binary):
        assert isinstance(unary, Operator) and unary.is_unary, "Must be an instance of 'Operator' and be unary"
        assert isinstance(binary, Operator) and not binary.is_unary, "must be an instance of 'Operator' and be binary"
        self.unary = unary
        self.binary = binary

    def __repr__(self):
        return "BothOperators({}, {})".format(self.unary, self.binary)
```

Stack and Queue

Implements the methods explained in design as well as the following Python-specific magic methods:

- **__bool__:** return whether or not it is NOT empty – the opposite of an 'is_full' method which is often implemented for stacks and queues.
- **__len__:** return the number of items in it.
- **__iter__:** return the items in it. This is used in Python in FOR loops so if I wrote the line 'for item in stack', it would call the '__iter__' method of 'stack' and set 'item' to each value of these in turn. However, I have implemented this as a generator as explained below to prevent unnecessary processing and memory usage.
- **__contains__:** return whether or not 'item' is in it.

I will implement the stacks and queues with the 'deque' library explained below which means most methods are interacting specifically with how this library works. This is another reason why I am making these methods – to hide the complexities of interacting with this. You could argue that it replaces the complexities of the deque with the complexities of my class so is not better, but my class follows the normal stack and queue methods that most programmers would know. This is also why I will not inherit from 'deque' because I want the signature of the stack and queue to follow what programmers would expect – push, pop, enqueue, dequeue and peek.

As I save the stack/queue as the attribute 'stack' or 'queue' respectively, I could just do 'for item in stack_instance.stack' but this is bad practice as the attribute 'stack' should be private to only allow the user to interact with it through methods. Therefore, I will make the attribute private so it can only be accessed from within the class.

Faster Access with deque

Stacks and queues are often implemented on static arrays which requires: all items to be the same type; the maximum length to be defined in advanced and never exceeded, so that this space can be reserved in memory; and the front of queues can't be filled without it being implemented as a circular queue. As I am programming in python which is dynamically typed and doesn't have arrays by default but lists, I will not be limiting the stacks or queues as above. This required more memory and time though so to make them more efficient, I will use a library to implement them:

Python has a library which has a class 'deque' which is optimised for interacting with items at the start and end of list-like objects. I could have built my Queue and Stack data structures from a list instead, but this way, accessing items at the front and back of the queue is done in constant time O(1) so if there were many, many operators, this would speed up the calculator. Realistically though, as the calculator will only be holding the tokens from the expression in the queue and stack, it will rarely be longer than 10 items so the effect will be negligible.

Generators

A generator is a non-greedy algorithm which means it does not have to have everything ready before it returns anything – it can prepare the first item, return it and the prepare the second, etc. This is important in algorithms because if I wanted to find the first 10 natural numbers, a greedy algorithm wouldn't work as it would try to prepare all natural numbers (of which there are infinite) before returning them, whereas a non-greedy algorithm will just prepare the first 10 and return them.

I will obviously not be working with infinite stacks/queues of operators but implementing the '__iter__' magic method as a generator means if the calculator errors after the first item in it, it will quit the loop and the other items won't even have been processed. In this way, it saves some processing time.

Code

```
class Stack:
    """Represents a stack - a LIFO data structure similar to a list/array with only access to the top"""

    def __init__(self):
        # private attribute denoted by the double underscore prefix
        self.__stack = deque()

    def push(self, item):
        """Add 'item' to the top of the stack"""
        self.__stack.append(item)

    def pop(self):
        """Remove and return the item on the top of the stack. Return 'None' if the stack is empty"""
        return self.__stack.pop() if self else None

    def peek(self):
        """Return the item on the top of the stack without removing it from the stack. Return 'None' if the stack is empty"""
        return self.__stack[-1] if self else None

    def __bool__(self):
        return bool(self.__stack)

    def __len__(self):
        return len(self.__stack)

    def __repr__(self):
        return "Stack({})".format(str(list(self.__stack)).replace("[", "").replace("]", ""))
```

```
    def __iter__(self):

        # need to copy it because the variable is a pointer to where the actual stack is so
        # creating a new variable will mean they both reference the same thing whereas now
        # they reference identical but different stacks which is what I want because I am removing
        # items from the copy which I don't want to happen to the real stack
        temp = self.__stack.copy()

        while temp:
            yield temp.pop()

    def __contains__(self, item):

        for i in self:
            if i == item:
                return True

        return False

class Queue:
    """Represents a queue - a FIFO data structure similar to a list/array with only access to the head and tail"""

    def __init__(self):
        # private attribute denoted by the double underscore prefix
        self.__queue = deque()

    def enqueue(self, item):
        """Add 'item' to the tail of the queue"""
        self.__queue.appendleft(item)

    def dequeue(self):
        """Remove and return the item at the head of the queue. Return 'None' if the queue is empty"""
        return self.__queue.pop() if self else None
```

```python
    def peek(self):
        """Return the item at the head of the queue without removing it from the queue. Return 'None' if the queue is empty"""
        return self.__queue[-1] if self else None

    def __bool__(self):
        return bool(self.__queue)

    def __len__(self):
        return len(self.__queue)

    def __repr__(self):
        return "Queue({})".format(str([item for item in self]).replace("[", "").replace("]", ""))

    def __iter__(self):

        # need to copy it because the variable is a pointer to where the actual queue is so
        # creating a new variable will mean they both reference the same thing whereas now
        # they reference identical but different queues which is what I want because I am removing
        # items from the copy which I don't want to happen to the real queue
        temp = self.__queue.copy()

        while temp:
            yield temp.pop()

    def __contains__(self, item):

        for i in self:
            if i == item:
                return True

        return False
```

The '__repr__' magic methods aren't very readable, but they simply do some string operations to display what is in the queue or stack for debugging purposes.

'tokenise'

Regex

This regex will match portions of the expression against one of the following and give it the name in blue, so I know what it has matched it to:

```python
# compile the regex pattern that will be used to check for tokens
regex = compile_regex(r"""
    (?P<whitespace>\s+)
    |(?P<number>(\d*\.)?\d+)
    |(?P<bracket>[()])
    |(?P<other>.)
""", VERBOSE)
```

- **Whitespace:** '\s+' matches 1 or more whitespace characters, e.g.: spaces, newlines, etc. Although I will ignore whitespace at the next stage, it is important not to match it as anything else, so I will make it its own category.
- **Number:**
    - **Optional:**
        - **'\d*':** '\d' matches any digit from 0 to 9 so it matches 0 or more digits (allow 0 of them to allow '.5' meaning '0.5')
        - **'\.':** A decimal point (has to be escaped with a backslash because '.' will match anything in regex when not escaped)
    - **'\d+':** 1 or more digits
- **Bracket:** '[()]' matches a single open bracket or a single close bracket
- **Other:** '.' matches any single character but as it is at the end, anything that matches the above will not match this. In normal use, this will normally be a symbol such as '+' or '*' but it could be a letter, or anything not already matched. This means that every part of any expression will always match to something.

're.VERBOSE' allows me to put it on multiple lines and include comments in the middle so it is clearer.

The order of these matters because if the string appears to match more than 1 of the patterns in the regex, it will assume it is the first. If the string then stops matching this pattern despite all of it matching a pattern further down in the regex, it will not switch to it. Instead it will assume the string up until this point matched the pattern it matched to and at the point it stops, it thinks it is a new token and matched it to a different pattern. This means, if the 'other' pattern was first, everything would match to this.

Identify

To identify the tokens, I need the helper function 'should_be_unary' which returns whether or not an operator should be unary depending on what the token before was identified as.

```
def should_be_unary(prev_token):
    """Return whether or not the current token should be a unary operator depending on the previous token"""

    # if it is the first token in the expression, it should
    if prev_token is None:
        return True

    # if it's an open bracket, it should
    if isinstance(prev_token, OpenBracket):
        return True

    # if it's an operator, it should if it's binary or (unary and right associative)
    if isinstance(prev_token, Operator):

        # if it is binary, it should
        if not prev_token.is_unary:
            return True

        # if it is unary and right associative, it should
        if not prev_token.is_left_associative:
            return True

    # if none of the others are true, it shouldn't
    return False
```

Then I can use this in the identification algorithm:

```python
def identify(name, value, prev_token):
    """Return an instance of a class to identify the token"""

    # if it's a number, convert my standard form notation into python's and make it an instance of 'Num'
    if name == "number":
        return Num(value)

    # if it's a bracket, make it an instance of one of my bracket classes
    if value == "(":
        return OpenBracket()
    if value == ")":
        return CloseBracket()

    # if it's in 'valid_tokens', it's a valid operator so:
    if value in valid_tokens:
        token = valid_tokens[value]

        # if it could be unary or binary, use the helper function to decide which and return that
        if isinstance(token, BothOperators):
            return token.unary if should_be_unary(prev_token) else token.binary

        # if it's a function, create a unique instance and return that
        if isinstance(token, FunctionType):
            return token.create(calculate)

        # otherwise just return it
        return token

    # otherwise, it is an invalid token so error
    raise CalcError("Invalid token: '{}'".format(value))
```

Getting tokens from regex

The helper function to find which key found a match:

```python
def find_matched_key(match):
    """Return the key which was matched"""

    # exactly 1 value in 'match' will not be 'None'
    # so this will always return exactly once
    for key in match:
        if match[key] is not None:
            return key
```

Again, this looks like it may not return anything but because of how the regex library works and that I have an ‘other’ pattern that will catch everything, the return statement will run exactly once whatever the ‘match’ parameter is.

Now the whole tokenise algorithm can be put together:

```python
def tokenise(expr):
    """Split the expression up into tokens and make them instances of classes to identify them"""

    # remove whitespace at the start or end of the expression and make lower case
    expr = expr.strip().lower()

    # initialise variables
    tokens = []
    pos = 0

    while pos < len(expr):

        # find a regex match with the expression 'pos' characters in
        match = regex.match(expr, pos)

        # adjust 'pos' to be the end of the last match
        # so the next iteration doesn't match the same characters
        pos = match.end()

        # convert to a dictionary with the named patterns as keys
        match = match.groupdict()

        # find which key has been matched
        key = find_matched_key(match)

        # ignore whitespace
        if key != "whitespace":

            # the previous token is the last token in the list 'tokens[-1]' but if the list is empty, it is 'None'
            tokens.append(identify(key, match[key], tokens[-1] if tokens else None))

    return tokens
```

‘convert’

This only needs 1 helper function which helps to decide whether or not the token at the top of the stack should be executed before the current token:

```python
def should_be_executed_first(top_of_stack_token, current_token):
    """Return whether or not the token at the top of the stack should be executed before the current token"""

    # the token at the top of the stack should be executed before the current token if 1 and 2:
    #   1) the token at the top of the stack is an operator
    #   2) if a or b:
    #       a) the operator at the top of the stack has better precedence than the current operator
    #       b) i and ii:
    #           i)  they have equal precedences
    #           ii) the operator at the top of the stack is left associative

    #      ------------------ 1 ------------------ and -------------------------------------------------------------------------------------------- 2 ------------------------------------------------------------------------------------
    #                                                    --------------------------- a -------------------------- or ---------------------------------------------------- b ------------------------------------------------
    #                                                                                                                ---------------------------- i ---------------------------- and ------------------- ii ---------------
    return isinstance(top_of_stack_token, Operator) and (top_of_stack_token.precedence < current_token.precedence or (top_of_stack_token.precedence == current_token.precedence and top_of_stack_token.is_left_associative))
```

Now I can use this in the covert algorithm:

```python
def convert(tokens):
    """Convert the tokens from infix notation to postfix notation (AKA reverse polish notation) by the shunting yard algorithm"""

    output_queue = Queue()      # we only ever add numbers or operators to the output queue
    operator_stack = Stack()    # we only ever add operators and open brackets to the operator stack

    for token in tokens:

        # add numbers to the output queue
        if isinstance(token, Num):
            output_queue.enqueue(token)

        # if it's an operator, add any operators on the stack that should be executed before
        # it (using 'should_be_executed_first') to the output queue and then add it to the stack
        elif isinstance(token, Operator):
            while should_be_executed_first(operator_stack.peek(), token):
                output_queue.enqueue(operator_stack.pop())
            operator_stack.push(token)

        # add open brackets to the operator stack
        elif isinstance(token, OpenBracket):
            operator_stack.push(token)

        # otherwise, it must be a close bracket so add everything from the operator stack to the output queue
        # until there is an open bracket, then remove this too but don't add it to the output queue
        else:
            while not isinstance(operator_stack.peek(), OpenBracket) and operator_stack.peek() is not None:
                output_queue.enqueue(operator_stack.pop())
            operator_stack.pop()

    # move everything on the stack to the output queue
    while operator_stack:
        output_queue.enqueue(operator_stack.pop())

    return output_queue
```

‘execute’

This doesn’t need any helper functions.

```python
def execute(queue):
    """
    Execute the tokens to get a final answer
    As in postfix notation, the first operator in the queue is the first operator to be executed
    so execute this with the operands repeatedly until all of them have been executed to get a final answer
    """

    stack = Stack()

    for token in queue:

        # if it's a number, push it to the stack
        if isinstance(token, Num):
            stack.push(token)

        # otherwise it must be an operator so pop its operands from the stack, execute it with them and add push the result to the stack
        else:

            # at least 1 operand is needed for all operators
            operands = [stack.pop()]

            # binary operators need another
            if not token.is_unary:
                operands.append(stack.pop())

            # the stack returns None if empty so if 'None' is in there, there are too few operands
            if None in operands:
                raise CalcError("Too few operands or too many operators")

            # execute the operator with its operands (reversed)
            token = token.execute(operands[::-1])

            # make it a 'Num' and push it to the stack
            stack.push(Num(token))

    # there should be exactly 1 number on the stack at the end - the answer
    # if not, there are too many operands or too few operators
    if len(stack) != 1:
        raise CalcError("Too many operands or too few operators")

    # the last item on the stack is the answer
    return str(stack.pop())
```

I convert the answer to a string so the user doesn't receive an object of my 'Num' class as they will not know how it works. 'str' is a built-in type and can still clearly show the answer. Most users will be using it in an interface anyway so it will be displayed as a string anyway and those importing it will need to convert it to a number before using it.

'calculate'

```python
def calculate(expr, debug=False):
    """
    Calculate the answer to 'expr'.
    If CalcError (or it's child CalcOperationError) has been raised,
    it is due to an invalid expression so needs to be caught and presented as an error message
    Any other exceptions are errors in the code

    :param expr (str): The expression to execute
    :param debug (bool): Whether or not to print out extra information to check for errors. Default: False
    :return ans (str): The answer to 'expr'
    """

    assert isinstance(expr, str), "param 'expr' must be a string"

    for func in [tokenise, convert, execute]:

        # execute each function with the result from the last
        expr = func(expr)

        # output the progress if debug is on
        if debug:
            print(str(func).split("function ")[1].split(" at ")[0] + ":", repr(expr))

    return expr
```

The 'debug' flag defaults to 'False' and will output the progress of the calculator after each function has executed to make it easier to test and identify errors – I may remove it at the end.

The items in the list to iterate over are the identifiers of the functions which are in the main code above. They are not followed by brackets so are not called yet but their identifiers are just saved to the list so when they are iterated over, they can be called.

The 'expr' variable will change types as the loop iterates to whatever the functions return so it won't be an expression anymore, but this is easier than calling each of them in turn and will work when I expand to having more functions here.

Testing Operators

I also need to write the code for the 'valid_tokens' dictionary which includes the testing operators explained in analysis:

```python
# create the tokens that can be used in the calculator with the classes above and the operations in 'Operations.py'
# the key is the symbol that will be in expressions and the value is an instance of one of the following classes:
# UnaryOperator, BinaryOperator, Operator, or BothOperators
valid_tokens = {
    "-": BothOperators(UnaryOperator("negative", op_neg, False), BinaryOperator("subtraction", op_sub, 4, True)),
    "!": UnaryOperator("factorial", op_factorial, True),
    "^": BinaryOperator("exponentiation", op_exp, 1, False)
}
```

The precedence and associativity for each of these operations is shown in design and the code for them is:

```python
def op_sub(x, y):
    """Return x subtract y"""

    # typical cases
    return x - y

def op_exp(x, y):
    """Return x to the power of y"""

    # invalid case
    if x == 0 and y == 0:
        raise CalcOperationError("0 to the power of 0 is undefined", "^", [x, y])

    # typical cases
    return x ** y

def op_neg(x):
    """Return the negative of x"""

    # typical cases
    return -x

def op_factorial(x):
    """Return x factorial"""

    # invalid cases
    if x < 0 or x % 1 != 0:
        raise CalcOperationError("Must be whole number and cannot be negative", "!", [x])

    # multiply all numbers between 1 and x together
    product = 1
    while x > 1:
        product *= x
        x -= 1

    return product
```

Also, I added the following code to the end of the file which is a quick interface for me to test with:

```
# only runs if the file is run directly (not if imported)
if __name__ == "__main__":

    # quick interface to test the calculator with debug on to check it's working
    # catches errors due to the user's input and displays them as error messages
    # repeats until the user enters an empty expression
    expression = input("\n>")
    while expression != "":
        try:
            print(calculate(expression))
        except CalcError as e:
            print(e)
        expression = input("\n>")
```

2) Interface (Memory)

Docstring

First, I will explain what this file is for and how programmers can use it in a user interface. This is the docstring of the file which programmers will be able to see:

```
"""
Contains the interface between a user interface and the calculator

User interfaces should use the calculator via this to record and provide access to memory by instantiating the 'Interface' class and:
- if the user wants to calculate the answer to an expression, use the 'calculate' method
- if the user wants to view instructions, use the 'instructions' attribute
- if the user wants to view memory, use the 'recent_memory' method
- if the user wants to clear memory, use the 'clear_memory' method
- if the user wants to insert a specific memory answer into their expression:
    1) get which memory item is being requested
    2) use the 'memory_item' method to get the original expression and answer of interest
    3) it may be best to re-calculate the answer using the original expression and the 'calculate' method
    4) insert the answer from memory or the re-calculated answer into the expression

Before calling the 'recent_memory' or 'memory_item' methods with a number from the user,
the interface should call 'len_memory' to check how many items are in memory and verify the number wanted
is a valid number and equal to or less than the number of items in memory. If not, display the relevant error message
If either of these methods are called with invalid parameters, they will raise IndexError
"""
```

Methods

As these methods will be called by programmers making an interface, I will check for any errors and raise them straight away with helpful error messages so that the programmer can realise and fix them as soon as possible.

Constructor

Create the empty memory list and save the instructions as private attributes. Despite instructions needing to be accessed by user interfaces, I don't want them to be able to change it (they can add to it within the user interface, but they shouldn't be able to change this version).

```
def __init__(self):

    # private attributes
    self.__memory = []
    self.__instructions = instructions
```

Instructions Property

A property is a method that acts like an attribute. It is called without brackets as an attribute would but to get the value, a method is called. This means it seems like an attribute, but it cannot be changed. For user interfaces to extend the instructions, they must make their own local version of the instructions copied from here and extend that.

```
@property
def instructions(self):
    """Return the instructions without being able to change it"""
    return self.__instructions
```

Calculate

Calculate calls the main calculator for the answer as normal but also inserts both this and the original expression into memory before returning the answer. If there is an error, it will exit immediately so won't save anything in memory (if I saved the expression in memory before getting the answer, in the event of an error, it wouldn't save the answer so there would only be an expression in memory).

```
def calculate(self, expr):
    """
    Calculate the answer to 'expr', storing the expression and answer in memory for later recall
    If CalcError (or it's child CalcOperationError) has been raised,
    it is due to an invalid expression so needs to be caught and presented as an error message
    Any other exceptions are errors in the code

    :param expr (str): The expression to execute
    :return ans (str): The answer to 'expr'
    """

    # calculate the answer with the calculator
    ans = calculate(expr)

    # add the expression and answer to the front of the list
    self.__memory.insert(0, (expr, ans))

    return ans
```

Len Memory

```
def len_memory(self):
    """
    Return the number of items in memory

    :return (int): The number of items in memory
    """

    return len(self.__memory)
```

This can be used in user interfaces to verify whether memory references are valid.

Memory Item

```
def memory_item(self, num_calculations_ago=1):
    """
    Retrieve an answer from memory along with the expression that resulted in it
    Will raise 'IndexError' if 'num_calculations_ago' isn't an integer between 1 and the number of items in memory

    :param num_calculations_ago (int): The item to retrieve from memory: 1 is the most recent calculation, ascending from there. Default: None
    :return (tuple): A 2-value tuple where the 0th index is the string expression and the 1st is the string answer
    """

    # invalid cases
    if not isinstance(num_calculations_ago, int):
        raise IndexError("Must be an integer")
    if self.len_memory() < num_calculations_ago:
        raise IndexError("There aren't that many items saved in memory")
    if num_calculations_ago < 1:
        raise IndexError("Must be greater than or equal to 1")

    # typical cases
    return self.__memory[num_calculations_ago - 1]
```

Recent Memory

```
def recent_memory(self, num_to_retrieve=None):
    """
    Retrieve a list of answers from memory along with the expressions that resulted in each of them
    Will raise IndexError if 'num_to_retrieve' isn't an integer greater than or equal to 1

    :param num_to_retrieve (int): The number of answers to retrieve. 'None' means all. Default: None
    :return (list): The memory items (most recent first) which are each a 2-value tuple where the 0th
                    index is the string expression and the 1st is the string answer
    """

    # make 'None' mean all and if there are less than asked for, just return the number available
    if num_to_retrieve is None or num_to_retrieve > self.len_memory():
        num_to_retrieve = self.len_memory()

    # invalid cases
    if not isinstance(num_to_retrieve, int):
        raise IndexError("Must be an integer (whole number")
    if num_to_retrieve < 0:
        raise IndexError("Must be greater than or equal to 0")

    # typical cases
    # the slice selects the first 'num_to_retrieve' items in the list 'self.__memory'
    return self.__memory[:num_to_retrieve]
```

Clear Memory

```
def clear_memory(self):
    """Clear the calculator's memory"""

    self.__memory.clear()
```

Instructions in main calculator

To first get the instructions, I will store them in a text file which is imported by the main calculator and saved as a global variable so it can be imported. When users add custom operations, they should add instructions for it to this which will then be built upon by every subsequent file.

'Instructions.txt':

```
Enter an expression to calculate the answer.

Operators are represented by a symbol and perform an operation on the numbers around them.
Binary operators have 2 numbers (1 either side of the symbol), whereas unary operators have
1 number (either left or right of the symbol).

Operators are executed using BODMAS - brackets, other (exponents and unary operators),
division and multiplication, addition and subtraction.

Binary Operators:
Subtraction: use '-' between 2 numbers to find the first subtract the second.
Exponentiation: use '^' between 2 numbers to find the first to the power of the second.

Unary Operators:
Negative: use '-' before a number to find the negative of it.
Factorial: use '!' after a positive whole number to find the product of all positive whole
numbers less than or equal to it.
```

Reading them in at the start of 'Calc.py':

```python
# instructions read from the text file for other files to import
with open("Instructions.txt", "r") as f:
    instructions = "".join(f.readlines())
```

Testing Interface

If there is a problem, I raise an error, so it stops running the rest of the code until the error is caught. Otherwise it would still run the code below and cause another error. As I raise CalcError which is being caught anyway, it just provides a quick way to exit the program.

```
# only runs if the file is run directly (not if imported)
if __name__ == "__main__":

    # instantiate the class
    calc = Interface()

    # extend instructions
    new_instructions = calc.instructions + "\n\nType 'memory' to view previous calculatio

    # quick user interface to test the calculator through the interface and memory access
    # repeats until the user enters an empty expression
    expression = input("\n>")
    while expression != "":

        # typing 'instructions' will output the general instructions
        if "instructions" in expression:
            print(new_instructions)

        # typing 'clear' will clear the memory
        elif "clear" in expression:
            calc.clear_memory()
            print("Memory cleared")

        # typing 'recent memory' will output all previous calculations
        elif "memory" in expression:
            for expr, ans in calc.recent_memory():
                print("{} = {}".format(expr, ans))

        else:
            try:
                # including 'ans' will replace it with the previous answer
                if "ans" in expression:
                    expression = expression.replace("ans", calc.memory_item()[1])

                # including 'Mx' will replace it with the xth previous answer
                while "M" in expression:
                    index = expression.split("M")[1].split(" ")[0]
                    try:
                        index = int(index)
                    except ValueError:
                        raise CalcError("Memory references must be a whole numbers")
                    else:
                        if index > calc.len_memory():
                            raise CalcError("Not enough items in memory")
                        elif index < 1:
                            raise CalcError("Memory references must be greater than or equal to 1")
                    expression = expression.replace("M{}".format(index), calc.memory_item(index)[1], 1)

                # calculate the answer
                print(calc.calculate(expression))

            # catch and output errors
            except CalcError as e:
                print(e)

        expression = input("\n>")
```

The instructions edit continues but not all of it fits in the screenshot. I just explain how to use the keywords in this testing interface as an example of how to add to the instructions.

3) Graphical User Interface

Pygame VS Tkinter Prototype

In analysis, I prototyped for both Tkinter and Pygame and all my users preferred Pygame so I will use this.

Tkinter looks much more formal and all text entry has to be done in little forms whereas I want the expression to be input in a big open space in the middle which I can do in Pygame.

Pygame Tools

Now I have decided to use Pygame, I will create a file with some constants, functions and classes which will help me when making the rest of the calculator. These are not specific to the calculator but could help any programmer writing a window in Pygame.

Colour Constants

I will define common colours I may want to use in the calculator in a dictionary with the colour name as the key and the RGB colour code in a 3-value tuple as the value. This makes it quicker to reference colours in the calculator and I can play around with the colour codes to get the best colours once at the start without needing to remember them later on.

```
COLOURS = {
    "black":    (0,    0,      0),
    "white":    (255,  255,    255),
    "grey":     (128,  128,    128),
    "red":      (255,  0,      0),
    "green":    (0,    200,    0),
    "blue":     (0,    0,      255),
    "yellow":   (255,  255,    0)
}
```

Format Text Function

In order to display text in an area with finite space, I need to split it into lines, so it doesn't overflow off the screen. There is also finite vertical space so only a limited number of lines will fit on the screen. If the text is too long to fit on all the lines there is room for, I need to end with an ellipsis or add a scrollable screen instead.

The format text function will take a string of text, the maximum number of characters per line, the maximum number of lines (optional, defaults to unlimited) and whether or not to allow breaks in the middle of words. It will then format the text accordingly so all text fits in the space. I can then use this function to format any text that could have a variable length – the input expression, error message, answer, memory items and instructions.

The error message won't overflow onto more than 3 lines so I have given it this much space; the expression and answer have 5 lines and any more than that and they should use multiple expressions referring to the last calculation using memory; the memory items have 3 lines but don't need to show the whole thing, only enough for the user to recognise it. However, the instructions need to be shown in full so I will allow an unlimited number of lines and incorporate a scrolling surface.

```python
def format_text(text, MAX_CHARS_PER_LINE, MAX_LINES=None, break_anywhere=False):
    """Format text into lines so they fit on the screen and if it exceeds the maximum number of lines, trail off..."""

    # if we allow breaking the string anywhere, we just treat the string as an array of characters
    if break_anywhere:
        words = text

    # otherwise we split the text into an array of words (however we haven't dealt with newlines yet)
    else:
        words = text.split(" ")

    # initialise the new 'lines' list and 'current_line' string as empty
    lines = []
    current_line = ""

    # for each word in the instructions:
    word_num = 0
    while (MAX_LINES is None or len(lines) < MAX_LINES) and len(words) > word_num:
        word = words[word_num]

        # while there is a newline character in the current word:
        while "\n" in word:

            # the line is the word before the newline character
            line = word.split("\n")[0]

            # if it can't fit on the current line, add the current line and it as 2 separate lines
            if len(line) + len(current_line) > MAX_CHARS_PER_LINE:
                lines.append(current_line)
                lines.append(line)

            # if it can fit, add the word (add a space between if we don't allow breaking words anywhere)
            else:
                if break_anywhere:
                    lines.append(current_line + line)
                else:
                    lines.append(current_line + " " + line)

            # the current line is empty as after a newline character we always start a new line when there is a newline character
            current_line = ""

            # then replace the 'word' variable with all characters after the first newline character and continue the loop
            word = "\n".join(word.split("\n")[1:])

        # if there is enough room to put the word on the current line, add it
        # otherwise, add it to a new line
        if len(word) + len(current_line) > MAX_CHARS_PER_LINE:
            lines.append(current_line)
            current_line = word
        else:
            # only add a space if we don't allow breaking anywhere
            if break_anywhere:
                current_line += word
            else:
                current_line += " " + word

        word_num += 1

    # if we have run out of room, trail off
    if MAX_LINES is not None and len(lines) == MAX_LINES:
        lines[-1] += "..."

    # otherwise, add the last line
    else:
        lines.append(current_line)

    return lines
```

Middle Box Function

A simple function to find the middle of a box where in Pygame, a ‘box’ is a 4-value tuple where:

- Index 0 is the x ordinate of the top-left corner of the box
- Index 1 is the y ordinate of the top-left corner of the box
- Index 2 is the width of the box
- Index 3 is the height of the box

This means to find the middle, the new x ordinate is the old x ordinate plus half the width, and the new y ordinate is the old y ordinate plus half the height. However, pixel values have to be integers, so I use div to round down after division.

I use this to automatically position text in the middle of buttons.

```python
def middle_box(box):
    """Return the middle position of 'box'"""
    return box[0] + box[2] // 2, box[1] + box[3] // 2
```

Draw Class

My drawer simply stores the Pygame surface to draw to and the font to draw text with as attributes and it implements methods to create a text or button object with the provided font on the provided surface. This class means the surface and font don't have to be passed every time but only once at the start of the program.

```python
class Draw:
    """Drawer object which facilitates creating buttons and text in pygame"""
    def __init__(self, display, font):
        self.__display = display
        self.__font = font
    def button(self, box, box_colour, text_message, text_size, text_colour):
        """Create a button"""
        return Button(box, box_colour, self.text(text_message, text_size, text_colour, middle_box(box)), self.__display)
    def text(self, text_message, size, colour, center_pos):
        """Create text"""
        return Text(text_message, size, colour, center_pos, self.__font, self.__display)
```

Text Class

Stores the message, size, colour, centre position, font and surface to draw text and implements a method to draw it and to change the message.

```python
class Text:
    """Text object facilitating drawing to the screen and changing the text message"""
    def __init__(self, text_message, size, colour, center_pos, font, display):
        self.__font = pg.font.Font(font, round(size))
        self.__colour = colour
        self.__center_pos = center_pos
        self.__display = display
        self.__surf = self.__font.render(str(text_message), True, colour)
        self.__rect = self.__surf.get_rect()
        self.__rect.center = center_pos
    def draw(self):
        """Draw the text to the display"""
        self.__display.blit(self.__surf, self.__rect)
    def edit_text_message(self, new_text_message):
        """Edit the text message"""
        self.__surf = self.__font.render(str(new_text_message), True, self.__colour)
        self.__rect = self.__surf.get_rect()
        self.__rect.center = self.__center_pos
```

Button Class

Stores the box, colour, text object and surface to draw a button and implements methods to draw the button, check whether or not a position lies within the button and change the message on the text object of the button. In this way, aggregation occurs between the Text and Button classes as every Button object contains a Text object, but other Text objects can exist without a button.

```python
class Button:
    """Button object facilitating drawing it and the text in it to the screen, checking whether a point lies within it and changing the text message on the box"""
    def __init__(self, box, colour, text, display):
        self.__box = box
        self.__colour = colour
        self.__text = text
        self.__display = display
    def draw(self):
        """Draw the button and text in it to the display"""
        pg.draw.rect(self.__display, self.__colour, self.__box)
        self.__text.draw()
    def is_within(self, mouse_pos):
        """Return whether or not the mouse is within the button"""
        return self.__box[0] <= mouse_pos[0] <= self.__box[0] + self.__box[2] and self.__box[1] <= mouse_pos[1] <= self.__box[1] + self.__box[3]
    def edit_text_message(self, new_text_message):
        """Edit the text message on the button"""
        self.__text.edit_text_message(new_text_message)
```

Window

The code in 'PygameTools.py' is useful for many Pygame programs but any code specific to the calculator, I will include in the 'Window' class. Along with the constructor and 'run', 'mode_normal', 'mode_instructions' and 'calculate' methods, I will include a few more helper methods. All attributes and methods apart from the 'run' method will be private so it is clear what to do (as they can't do anything else) – instantiate the class and call the 'run' method to start the window.

Constructor

```python
def __init__(self):

    # constants
    self.__RESOLUTION = self.__WIDTH, self.__HEIGHT = 800, 600
    self.__TARGET_FPS = 30
    self.__BACKGROUND_COLOUR = COLOURS["grey"]
    self.__FONT = "freesansbold.ttf"

    # variables needed for the calculator
    self.__calculator = Interface()
    self.__expr = self.__ans = self.__error_msg = ""

    # the current mode name and mapping between mode names and their methods
    self.__mode = "normal"
    self.__modes = {
        "normal": self.__mode_normal,
        "instructions": self.__mode_instructions
    }

    # the distance the instructions scrollable view is positioned above the top of the screen
    self.__scroll = 0

    # whether or not to exit the calculator
    self.__done = False
```

Run Method

Unfortunately, I cannot do all the processing I would like to do in the constructor because for a lot of it, I need the Pygame window to be open. Once I have done this, I can create my drawer from 'PygameTools.py' and start creating all the text and button objects I need. Despite these not being in the constructor, these are all done outside the main loop so it should have little processing to do every tick.

I need extra text objects for the memory as only 1 line of text can be inside a button but as I am allowing 3 lines per memory item, if it overflows, I need more text objects that aren't associated with a button.

```python
def run(self):
    """Run the calculator user interface"""

    # start pygame, create the window, caption it and start the clock
    pg.init()
    self.__display = pg.display.set_mode(self.__RESOLUTION)
    pg.display.set_caption("Calculator")
    clock = pg.time.Clock()

    # create my drawer for drawing things on the screen
    self.__drawer = Draw(self.__display, self.__FONT)

    # create buttons and text I will draw later
    self.__button_instructions = self.__drawer.button((0, 0, 200, 50), COLOURS["yellow"], "Instructions", 25, COLOURS["black"])
    self.__button_clear_memory = self.__drawer.button((200, 0, 200, 50), COLOURS["red"], "Clear Memory", 25, COLOURS["black"])
    self.__button_back = self.__drawer.button((self.__WIDTH - 100, 0, 100, 50), COLOURS["black"], "Back", 25, COLOURS["white"])
    self.__texts_expr = []
    self.__texts_ans = []
    self.__texts_error_msg = []
    self.__text_instructions_title = self.__drawer.text("Instructions", 25, COLOURS["yellow"], (400, 50))
    self.__text_memory = self.__drawer.text("Memory", 25, COLOURS["green"], (700, 50))
    self.__buttons_memory = []
    self.__text_extra_memory = []
    self.__format_instructions()

    # main loop
    while not self.__done:

        # clear screen
        self.__display.fill(self.__BACKGROUND_COLOUR)

        # get events
        events = pg.event.get()

        # let windows close the window
        for event in events:
            if event.type == pg.QUIT:
                self.__done = True

        if not self.__done:

            # call the current mode's method
            self.__modes[self.__mode](events)

            # update the display and tick the clock
            pg.display.update()
            clock.tick(self.__TARGET_FPS)

    # close the pygame window
    pg.quit()
```

It uses the private helper method ‘format_instructions’ below:

Format Instructions

In order to fit all the instructions on the screen, I will draw the instructions onto an intermediate surface which I can draw in different positions depending on how much the user has scrolled.

The extension of the instructions continues but not all of it fits on the screenshot.

```python
def __format_instructions(self):
    """Format the instructions into lines on a scrollable surface"""

    # extend the instructions
    instructions = "SCROLL DOWN TO VIEW MORE:\n\n" + self.__calculator.instructions + "\n\nTo insert a previous answer

    # format the instructions into lines
    lines = format_text(instructions, 75)

    # make the intemediate surface just bit enough to hold all the instruction lines and make it the background colour
    self.__intermediate = pg.surface.Surface((self.__WIDTH, len(lines) * 20 + 10))
    self.__intermediate.fill(self.__BACKGROUND_COLOUR)

    # make a new drawer to draw on the intemediate surface
    new_drawer = Draw(self.__intermediate, self.__FONT)

    # create a text object for each line
    self.__texts_instructions = []
    count = 0
    for line in lines:
        self.__texts_instructions.append(new_drawer.text(line, 20, COLOURS["black"], (400, 10 + 20 * count)))
        count += 1
```

Update Text and Buttons

Rather than creating and drawing new objects every tick, I create objects outside the main loop and only change or add then when I know they could change. This function takes Boolean parameters for each of these and updates the ones that need to be updated.

```python
def __update_text_and_buttons(self, memory=False, expr=False, ans=False, error=False):
    """
    Update the message on text and button objects if the message has changed
    The parameters are whether or not to update the message on those text/button objects
    """

    # if we need to update the memory buttons:
    if memory:

        self.__text_extra_memory = []
        count = 0

        # for the most recent 5 items in memory:
        for expression, answer in self.__calculator.recent_memory(5):

            # format the text for each memory item into lines
            lines = format_text("{}: {} ({})".format(count + 1, answer, expression), 15, 3, True)

            # if there is only 1 line, make it
            if len(lines) == 1:

                # if there is already a button, change the text
                if len(self.__buttons_memory) > count:
                    self.__buttons_memory[count].edit_text_message(lines[0])

                # otherwise create a button with the new text
                else:
                    self.__buttons_memory.append(self.__drawer.button((600, 100 * (count + 1), 200, 100), COLOURS["black"], lines[0], 20, COLOURS["white"]))

            # otherwise make it and add the other 1 or 2 lines
            else:
                # if there is already a button, change the text
                if len(self.__buttons_memory) > count:
                    self.__buttons_memory[count].edit_text_message(lines[1])

                # otherwise create a button with the new text
                else:
                    self.__buttons_memory.append(self.__drawer.button((600, 100 * (count + 1), 200, 100), COLOURS["black"], lines[1], 20, COLOURS["white"]))

                # add the first line
                self.__text_extra_memory.append(self.__drawer.text(lines[0], 20, COLOURS["white"], (700, (100 * count) + 125)))

                # if there are 3 lines, add this too
                if len(lines) == 3:
                    self.__text_extra_memory.append(self.__drawer.text(lines[2], 20, COLOURS["white"], (700, (100 * count) + 175)))

            count += 1

        # remove buttons that aren't in memory anymore
        self.__buttons_memory = self.__buttons_memory[:count]
    # if we need to update the expression text objects:
    if expr:

        # format the expression into lines
        lines = format_text(self.__expr, 18, 5, True)
        count = 0
        for line in lines:

            # if there are already enough objects, change the message on it
            if len(self.__texts_expr) > count:
                self.__texts_expr[count].edit_text_message(line)

            # otherwise, create a new object with the message
            else:
                self.__texts_expr.append(self.__drawer.text(line, 35, COLOURS["blue"], (300, 200 + (30 * count))))

            count += 1

        # remove unnecessary objects
        self.__texts_expr = self.__texts_expr[:count]

    # if we need to update the answer text objects:
    if ans:

        # format the answer into lines
        lines = format_text(self.__ans, 18, 5, True)
        count = 0
        for line in lines:

            # if there are already enough objects, change the message on it
            if len(self.__texts_ans) > count:
                self.__texts_ans[count].edit_text_message(line)

            # otherwise, create a new object with the message
            else:
                self.__texts_ans.append(self.__drawer.text(line, 35, COLOURS["green"], (300, 400 + (30 * count))))

            count += 1

        # remove unnecessary objects
        self.__texts_ans = self.__texts_ans[:count]
```

```
# if we need to update the error text objects:
if error:

    # format the error message into lines
    lines = format_text(self.__error_msg, 50, 2)
    count = 0
    for line in lines:

        # if there are already enough objects, change the message on it
        if len(self.__texts_error_msg) > count:
            self.__texts_error_msg[count].edit_text_message(line)

        # otherwise, create a new object with the message
        else:
            self.__texts_error_msg.append(self.__drawer.text(line, 25, COLOURS["red"], (300, 100 + (25 * count))))

        count += 1

    # remove unnecessary objects
    self.__texts_error_msg = self.__texts_error_msg[:count]
```

Normal Mode

Runs every tick when in normal mode. The line that overflows the screenshot is simply a list of all numeric characters on the keyboard – 0-9 on the numbers above the letters and 0-9 on the key pad.

```
def __mode_normal(self, events):
    """Runs every tick when in normal mode"""

    # draw buttons and text
    self.__button_instructions.draw()
    self.__button_clear_memory.draw()
    self.__text_memory.draw()
    for button in self.__buttons_memory:
        button.draw()
    for text in self.__texts_expr + self.__texts_ans + self.__texts_error_msg + self.__text_extra_memory:
        text.draw()

    # draw lines between the buttons to distinguish them (same colour as background so when they aren't there it looks the same)
    for count in range(2, 5+1):
        pg.draw.line(self.__display, self.__BACKGROUND_COLOUR, (600, 100 * count), (800, 100 * count))

    # handle events
    for event in events:
        if event.type == pg.KEYDOWN:

            # if the user pressed escape, clear the expression
            if event.key == pg.K_ESCAPE:
                self.__expr = ""
                self.__update_text_and_buttons(expr=True)

            # if the user presses backspace, remove 1 character from the expression
            elif event.key == pg.K_BACKSPACE:
                self.__expr = self.__expr[:-1]
                self.__update_text_and_buttons(expr=True)

            # if either or the return/enter keys are pressed, call the 'calculate' method
            elif event.key == pg.K_RETURN or event.key == pg.K_KP_ENTER:
                self.__calculate()

            # otherwise add the text to the expression
            else:

                # if the first thing they type isn't a number, insert the last answer into the start of the expression
                if event.key not in [pg.K_0, pg.K_1, pg.K_2, pg.K_3, pg.K_4, pg.K_5, pg.K_6, pg.K_7, pg.K_8, pg.K_9, pg.K_KP0, pg.K_KP1, pg.K_KP2,
                    self.__expr = self.__ans

                self.__ans = self.__error_msg = ""
                self.__expr += event.unicode
                self.__update_text_and_buttons(expr=True, ans=True, error=True)
        elif event.type == pg.MOUSEBUTTONDOWN and event.button == 1:
            mouse_pos = pg.mouse.get_pos()

            # if the user clicked the instructions button, change to instructions mode
            if self.__button_instructions.is_within(mouse_pos):
                self.__mode = "instructions"

            # if the user clicked the clear memory button, clear the memory
            elif self.__button_clear_memory.is_within(mouse_pos):
                self.__calculator.clear_memory()
                self.__ans = self.__error_msg = ""
                self.__update_text_and_buttons(memory=True, ans=True, error=True)

            # if the user clicked on a memory item, insert that item into the expression
            else:
                for button in self.__buttons_memory:
                    if button.is_within(mouse_pos):
                        self.__expr += "M{}".format(self.__buttons_memory.index(button) + 1)
                        self.__update_text_and_buttons(expr=True)
```

There is a lot to do when the user pressed ENTER on an expression to calculate it, so I have split it out into another method ‘calculate’:

Calculate

```python
def __calculate(self):
    """Calculate the answer to the expression and update everything"""

    # remove whitespace at the start and end, make lower case and replace 'ans' with 'm1'
    expr = self.__expr.strip().lower().replace("ans", "m1")

    # replace all memory references with the actual answers and call the main calculator
    # with the resulting expression, catching errors and displaying them
    try:
        self.__ans = self.__calculator.calculate(self.__convert_memory_references(expr))
    except CalcError as e:
        self.__error_msg = str(e)
        self.__ans = ""
        self.__update_text_and_buttons(ans=True, error=True)
    else:
        self.__error_msg = ""
        self.__expr = ""
        self.__update_text_and_buttons(True, True, True, True)
```

This in turn needs to replace all memory references with the actual answer which I split out into another method:

Convert Memory References

```python
def __convert_memory_references(self, expr):
    """Convert all memory references to the actual answers"""

    # runs for every 'm' in 'expr'
    for _ in range(expr.count("m")):

        # get the string after the first 'm'
        index = expr.split("m")[1]

        # find the number of characters until the first non-numerical character
        pos = 0
        while pos < len(index) and index[pos] in "0123456789":
            pos += 1

        # find the string between the first 'm' and the first non-numerical characters
        # the slice means all characters up to pos characters through it
        index = index[:pos]

        # if there is a number after the 'm':
        if index != "":
            index = int(index)

            # if valid, replace the reference with the actual answer
            if 1 <= index <= self.__calculator.len_memory():
                expr = expr.replace("m{}".format(index), "(" + self.__calculator.memory_item(index)[1] + ")", 1)

            # otherwise, give an error message
            else:
                raise CalcError("Memory references must be between 1 and the number of items in memory")

    return expr
```

The counter ‘pos’ only increases if the next character is numerical so I could be left with an empty string in ‘index’. If this is the case, assume the ‘m’ is part of something else that the main calculator may understand and if it doesn’t it can error there. However, if there is a number after the ‘m’ but it is invalid, raise an error now.

Instructions Mode

```python
def __mode_instructions(self, events):
    """Runs every tick when in instructions mode"""

    # draw the instructions text onto the intemediate surface
    for line in self.__texts_instructions:
        line.draw()

    # draw the intemediate surface onto the main surface
    self.__display.blit(self.__intermediate, (0, 100 + self.__scroll))

    # draw a rectangle, the title and back button over the top of the top of the surface
    pg.draw.rect(self.__display, self.__BACKGROUND_COLOUR, (0, 0, self.__WIDTH, 100))
    self.__text_instructions_title.draw()
    self.__button_back.draw()

    # handle events
    for event in events:

        # if the user pressed escape or clicked the back button, change to normal mode
        if event.type == pg.KEYDOWN:
            if event.key == pg.K_ESCAPE:
                self.__mode = "normal"
        elif event.type == pg.MOUSEBUTTONDOWN:
            if event.button == 1:
                mouse_pos = pg.mouse.get_pos()
                if self.__button_back.is_within(mouse_pos):
                    self.__mode = "normal"

            # if the user scrolled up, lower the intemediate surface but not lower than the origin
            elif event.button == 4:
                self.__scroll = min(self.__scroll + 15, 0)

            # if the user scrolled down, raise the intemediate surface but not higher than the point
            # where the bottom instructions line is fully visible
            elif event.button == 5:
                self.__scroll = max(self.__scroll - 15, self.__HEIGHT - len(self.__texts_instructions) * 20 - 100)
```

Here I edit the ‘scroll’ variable when the user scrolls so I can adjust the position of the intermediate surface.

Files

The user interface is split into 2 files – 1 file with the Pygame tools in ‘PygameTools.py’ and the other with the ‘Window’ class in ‘UserInterface.pyw’. I have made it a ‘.pyw’ so it doesn’t open the python console when running as I don’t print anything and the Pygame window is all I need.

PygameTools.py

```python
"""
Tools for writing pygame windows
"""

import pygame as pg

COLOURS = {
    "black":    (0,    0,     0),
    "white":    (255,  255,   255),
    "grey":     (200,  200,   200),
    "red":      (255,  0,     0),
    "green":    (0,    200,   0),
    "blue":     (0,    0,     255),
    "yellow":   (255,  255,   0)
}

def format_text(text, MAX_CHARS_PER_LINE, MAX_LINES=None, break_anywhere=False): ...

def middle_box(box): ...

class Text: ...

class Button: ...

class Draw: ...
```

66

UserInterface.pyw

```python
"""
A graphical user interface for the calculator
To open, instantiate the 'Window' class and then call the 'run' method
"""

import pygame as pg
from PygameTools import COLOURS, Draw, format_text
from Interface import Interface
from Errors import CalcError

class Window:...

if __name__ == "__main__":
    Window().run()
```

4) Constants

I added the testing constants – 'pi' and 'e' to 29 decimal places so the answer should be accurate to 28 decimal places as the calculator can display. They have to be strings to be represented accurately – otherwise they are stored in floating point binary which cannot accurately represent all numbers before being converted to a 'Decimal'.

The identification algorithm simply checks whether anything else found is in the 'valid_tokens' dictionary so this doesn't need to be changed, however, the regex splits anything else up into individual characters so 'pi' becomes 'p' and 'i' which are (individually) not in 'valid_tokens'. To fix this, I need to create a new pattern in the regex to keep words together. This is easy enough in regex – '[a-z]+' matches 1 or more lower case letters and as I lower the input string before processing, this will work for all letters.

```python
# create the tokens that can be used in the calculator with the classes above and the operations in 'Operations.py'
# the key is the symbol that will be in expressions and the value is an instance of one of the following classes:
# UnaryOperator, BinaryOperator, Operator, or BothOperators
valid_tokens = {
    "-": BothOperators(UnaryOperator("negative", op_neg, False), BinaryOperator("subtraction", op_sub, 4, True)),
    "!": UnaryOperator("factorial", op_factorial, True),
    "^": BinaryOperator("exponentiation", op_exp, 2, False),
    "pi": Num("3.14159265358979323846264338327"),
    "e": Num("2.71828182845904523536028747135")
}

# compile the regex pattern that will be used to check for tokens
regex = compile_regex(r"""
    (?P<whitespace>\s+)
    |(?P<number>(\d*\.)?\d+)
    |(?P<word>[a-z]+)
    |(?P<bracket>[()])
    |(?P<other>.)
""", VERBOSE)
```

5) Standard Form

Regex

In order to match numbers in standard form, more regex is required. To turn a number into standard form, you just add '~' to the end of it then either a '+', a '-', or nothing followed by the integer exponent. The regex for this is '**~[+-]?\d+**':

- **'~':** A single tilde character meaning 'times 10 to the …'
- **'[+-]?':** Either nothing or a single plus sign or a single minus sign
- **'\d+':** 1 or more digits from 0 to 9 (the exponent)

As not all numbers have to be in standard form, this is optional so that whole section is surrounded by brackets and followed by a question mark and it can go on the end of our previous number pattern:

```python
# compile the regex pattern that will be used to check for tokens
regex = compile_regex(r"""
    (?P<whitespace>\s+)
    |(?P<number>(\d*\.)?\d+(~[+-]?\d+)?)
    |(?P<word>[a-z]+)
    |(?P<bracket>[()])
    |(?P<other>.)
""", VERBOSE)
```

Conversions

To convert '~' to 'e', in 'identify', if it matched to the 'number' pattern, I can replace any '~'s in the expression with 'e':

```
# if it's a number, make it an instance of 'Num'
if name == "number":

    # convert my standard form notation into python's
    value = value.replace("~", "e")

    return Num(value)
```

To convert back, in 'execute', I need to lower the output string too because normal python uses a lower case 'e' in standard form whereas the 'decimal' library uses upper case 'E':

```
# the last item on the stack is the answer but we need to convert back to '~'
return str(stack.pop()).lower().replace("e", "~")
```

6) Functions

Datatypes

```
class FunctionType:
    """
    Represents a type of function and stores information about it

    :param name (str): The name of the type of function
    :param func (identifier): The identifier of the function to execute the operation
    """

    def __init__(self, name, func):
        self.__name = name
        self.__func = func

    def create(self, calc):
        """
        Return a new object which has the same properties as this object
        but is unique for all instances of the function in the expression

        :param calc (function): The calculate function from the main calculator
        :return (object): An instance of the 'FunctionInstance' class
        """

        return FunctionInstance(self.__name, self.__func, calc)

    def __repr__(self):
        return "FunctionType({})".format(self.__name)
```

```python
class FunctionInstance:
    """
    Represents a function instance and stores information about it

    :param name (str): The name of the type of function
    :param func (function): The function to execute the operation
    :param calc (function): The calculate function from the main calculator
    """

    def __init__(self, name, func, calc):
        self.__name = name
        self.__func = func
        self.__calc = calc
        self.__operands = []

    def add_operand(self, operand):
        """
        Execute the operand with the calculator to simplify it and then add it to the stored operands

        :param operand (num): The operand to add
        """

        self.__operands.append(Num(self.__calc(operand)))

    def execute(self):
        """
        Recursively execute all operands using the calculator and then
        return the answer when the function is executed with its operands

        :return (Num): The answer to the function when executed on its operands
        """

        # execute the function with its operands and return it
        # the star splits the list out into individual arguments
        return Num(self.__func(*self.__operands))

    def __repr__(self):
        return "{}({})".format(self.__name, ", ".join([str(operand) for operand in self.__operands]))
```

Regex

The comma pattern has been added:

```python
# compile the regex pattern that will be used to check for tokens
regex = compile_regex(r"""
    (?P<whitespace>\s+)
    |(?P<number>(\d*\.)?\d+(~[+-]?\d+)?)
    |(?P<word>[a-z]+)
    |(?P<bracket>[()])
    |(?P<comma>,)
    |(?P<other>.)
""", VERBOSE)
```

Identify

If a comma is found, I error and if a function is found, I create a new instance unique to every function in the expression. I pass the calculate function from the main calculator to this so the operands can be executed recursively.

```python
def identify(name, value, prev_token):
    """Return an instance of a class to identify the token"""

    # if it's a number, convert my standard form notation into python's and make it an instance of 'Num'
    if name == "number":
        return Num(value.replace("~", "e"))

    # if it's a bracket, make it an instance of one of my bracket classes
    if value == "(":
        return OpenBracket()
    if value == ")":
        return CloseBracket()

    # if it's a comma, give an error message as commas can only be inside functions
    if value == ",":
        raise CalcError("Commas only allowed inside functions")

    # if it's in 'valid_tokens', it's a valid operator so:
    if value in valid_tokens:
        token = valid_tokens[value]

        # if it could be unary or binary, use the helper function to decide which and return that
        if isinstance(token, BothOperators):
            return token.unary if should_be_unary(prev_token) else token.binary

        # if it's a function, create a unique instance and return that
        if isinstance(token, FunctionType):
            return token.create(calculate)

        # otherwise just return it
        return token

    # otherwise, it is an invalid token so error
    raise CalcError("Invalid token: '{}'".format(value))
```

Tokenise

The new status variable ‘in_func’ has been defined at the top and then the only changed occur inside the if/else block checking ‘in_func’.

If not ‘in_func’, I do what I did previously, but if the token is a function, I turn ‘in_func’ on.

However, if it is on, I do what I explained in design. The bracket depth can only be 0 after a close bracket so I only check for this there.

```python
def tokenise(expr):
    """Split the expression up into tokens and make them instances of classes to identify them"""

    # remove whitespace at the start or end of the expression and make lower case
    expr = expr.strip().lower()

    # initialise variables
    tokens = []
    pos = 0
    in_func = False

    while pos < len(expr):

        # find a regex match with the expression 'pos' characters in
        match = regex.match(expr, pos)

        # adjust 'pos' to be the end of the last match
        # so the next iteration doesn't match the same characters
        pos = match.end()

        # convert to a dictionary with the named patterns as keys
        match = match.groupdict()

        # find which key has been matched
        key = find_matched_key(match)

        # ignore whitespace
        if key != "whitespace":

            # if not in a function, identify the token and add it to the list of tokens
            if not in_func:

                # the previous token is the last token in the list 'tokens[-1]' but if the list is empty, it is 'None'
                token = identify(key, match[key], tokens[-1] if tokens else None)
                tokens.append(token)

                # if the token is a function, initialise the function variables
                if isinstance(token, FunctionInstance):
                    in_func = True
                    bracket_depth = 0
                    operand_start_pos = pos
            # if in a function, ignore everything except brackets and commas
            # to get the operands as strings and add them to the function object
            else:
                if key == "bracket":

                    # if it's an open bracket, increase the bracket depth
                    if match[key] == "(":
                        bracket_depth += 1

                    # if it's a close bracket, decrease the bracket depth
                    else:
                        bracket_depth -= 1

                        # if the bracket depth is now 0, add the last operand and execute the function
                        if bracket_depth == 0:
                            in_func = False
                            identify_operand(expr[operand_start_pos:pos], tokens[-1])
                            tokens[-1] = tokens[-1].execute()

                # if it's a comma, add the operand to the function
                # and update the start pos for the next operand
                elif key == "comma" and bracket_depth == 1:
                    identify_operand(expr[operand_start_pos:pos], tokens[-1])
                    operand_start_pos = pos - 1

                # add omitted closing brackets around functions
                elif pos >= len(expr):
                    expr += ")"

    return tokens
```

Identify Operand

The new helper function I created to identify and add the operand to the function object. The error messages only reference lack of brackets at the start or end of the function as if a comma was emitted in the middle of the function, it would just count the 2 operands either side of where the comma should be as 1 operand and probably error somewhere else.

```
def identify_operand(operand, function):
    """Validate and format the operand of a function and store it in the function"""

    #remove whitespace at the start and end of the operand
    operand = operand.strip()

    # remove the '(' or ',' at the start of the operand or error if neither
    if operand[0] == "(" or operand[0] == ",":
        operand = operand[1:]
    else:
        raise CalcError("Functions must be immediately followed by brackets")

    # remove the ',' or ')' at the end of the operand or error if neither
    if operand[-1] == ")" or operand[-1] == ",":
        operand = operand[:-1]
    else:
        raise CalcError("Functions must end with a close bracket")

    # add the operand to the function object
    function.add_operand(operand)
```

Testing Functions

The 3 testing functions have been added to 'valid_tokens':

```
# create the tokens that can be used in the calculator with the classes above and the operations in 'Operations.py'
# the key is the symbol that will be in expressions and the value is an instance of one of the following classes:
# UnaryOperator, BinaryOperator, Operator, or BothOperators
valid_tokens = {
    "-": BothOperators(UnaryOperator("negative", op_neg, False), BinaryOperator("subtraction", op_sub, 4, True)),
    "!": UnaryOperator("factorial", op_factorial, True),
    "^": BinaryOperator("exponentiation", op_exp, 2, False),
    "pi": Num("3.14159265358979323846264338327"),
    "e": Num("2.71828182845904523536028747135"),
    "sin": FunctionType("sin", func_sin),
    "root": FunctionType("root", func_root),
    "sum": FunctionType("sum", func_sum)
}
```

The code to execute them has been added to 'Operations.py':

```
def func_sin(x):
    """Return sin(x) where x is in radians"""

    # use the function from the 'math' library
    return sin(x)
```

```python
def func_root(root, x):
    """Return the root th root of x"""

    # invalid cases
    if root <= 0 or root % 1 != 0:
        raise CalcOperationError("The root must be a positive whole number", "¬", [root, x])

    # specific case of power
    return x ** (1 / root)
```

```python
def func_sum(x, y, z):
    """Return the sum of x, y and z"""

    return x + y + z
```

7) Operations

To add the operations I designed, I just have to add their details to ‘valid_tokens’, write the code for them and describe how to use them in the instructions. Many operations are built-in to Python so I will use these if available. I will write my own algorithms for most of the rest. The operations I will not write my own algorithms for are logarithms, circular and hyperbolic functions, and random number generation because they require tables of data and truly random data. I will use the ‘math’ and ‘random’ libraries for these instead.

Whatever the algorithm, I will always check for invalid domains and raise my custom error messages so that I will catch and present them to the user. This also means I can make the error messages as intuitive as possible.

Instructions

I added details of how to use each type of operator, function and constant to ‘Instructions.txt’:

```
Enter an expression to calculate the answer.

Operators are represented by a symbol and perform an operation on the numbers around them. Binary operators have 2 numbers (1 either side of the symbol), whereas unary operators have 1 number (either left or right of the symbol). Functions are represented by a word followed by brackets containing all operands (values needed for the function) separated by commas. Constants are a word representing a number very accurately. Simply enter the word and it will convert it to the number.

Functions and constants are always executed first and then operators are executed using BODMAS - brackets, other (exponents and unary operators), division and multiplication, addition and subtraction.

Binary Operators:
Addition: use '+' between 2 numbers to find the first add the second.
Subtraction: use '-' between 2 numbers to find the first subtract the second.
Multiplication: use '*' between 2 numbers to find the first multiplied by the second.
Division (true): use '/' between 2 numbers to find the first divided by the second.
Division (floor): use '\' between 2 numbers to find the first divided by the second and rounded down to the nearest whole number.
Mod: use '%' between 2 numbers to find the remainder after the first is divided by the second.
Exponentiation: use '^' between 2 numbers to find the first to the power of the second.
Root: use '¬' to find the first th root of the second - '2¬a' is the square root of 'a', '3¬a' is the cube root of 'a', etc.
Permutations: use 'P' to find the number of ways there are to organise the second number of items into the first number of places including all possible orders.
Combinations: use 'C' to find the number of ways there are to organise the second number of items into the first number of places only counting 1 possible order.

Unary Operators:
Positive: use '+' before a number to find the positive of it.
Negative: use '-' before a number to find the negative of it.
Factorial: use '!' after a positive whole number to find the product of all positive whole numbers less than or equal to it.
```

```
Functions:
Natural log: use 'ln' with 1 operand to find the natural logarithm of it.
Logarithm: use 'log' with 2 operands to find the logarithm of the first to the second base.
Absolute value: use 'abs' with 1 operand to find the absolute value of it which is always positive.
Lowest common multiple: use 'lcm' with 2 operands to find the lowest common multiple of them.
Highest common factor: use 'hcf' with 2 operands to find the highest common factor of them.
Random number generator: use 'rand' with 2 operands to find a random integer between them, inclusive.
Quadratic equation solver: use 'quadp' with 3 operands (a, b and c) to find the positive square root answer to the quadratic equation
'ax^2 + bx + c = 0' or use 'quadn' to find the negative square root answer of the same equation.
Sine: use 'sin' with 1 operand (an angle) to find the ratio between the opposite side and hypotenuse of its triangle.
Cosine: use 'cos' with 1 operand (an angle) to find the ratio between the adjacent side and hypotenuse of its triangle.
Tangent: use 'tan' with 1 operand (an angle) to find the radio between the opposite and adjacent sides of its triangle.
Inverse sine: use 'arsin' with 1 operand between -1 and 1 inclusive to find the angle it makes with the opposite side and hypotenuse of
its triangle.
Inverse cosine: use 'arcos' with 1 operand between -1 and 1 inclusive to find the angle it makes with the adjacent side and hypotenuse
of its triangle.
Inverse tangent: use 'artan' with 1 operand to find the angle it makes with the opposite and adjacent sides of its triangle.
Hyperbolic sine: use 'sinh' with 1 operand (an angle) to find the ratio between the opposite side and hypotenuse of its hyperbola.
Hyperbolic cosine: use 'cosh' with 1 operand (an angle) to find the ratio between the adjacent side and hypotenuse of its hyperbola.
Hyperbolic tangent: use 'tanh' with 1 operand (an angle) to find the ratio between the opposite and adjacent sides of its hyperbola.
Inverse hyperbolic sine: use 'arsinh' with 1 operand to find the angle it makes with the opposite side and hypotenuse of its hyperbola.
Inverse hyperbolic cosine: use 'arcosh' with 1 operand at least 1 to find the angle it makes with the adjacent side and hypotenuse of
its hyperbola.
Inverse hyperbolic tangent: use 'artanh' with 1 operand between -1 and 1 inclusive to find the angle it makes with the opposite and
adjacent sides of its hyperbola.

Constants:
pi: use 'pi' to get the ratio between a circle's circumference and its diameter. Value = 3.1415...
tau: use 'tau' to get 2 lots of pi - the number of radians in 360 degrees. Value = 6.2831...
e: use 'e' to get Euler's number. Value = 2.7182...
g: use 'g' to get the acceleration due to gravity close to the Earth's surface. Value = 9.80665
phi: use 'phi' to get the golden ratio found in many places in nature. Value = 1.6180...
```

Valid Tokens

I added each operator, function and constant's details to the 'valid_tokens' dictionary in 'Datatypes.py':

```
# create the tokens that can be used in the calculator with the classes above and the operations in 'Operations.py'
# the key is the symbol that will be in expressions and the value is an instance of one of the following classes:
# UnaryOperator, BinaryOperator, Operator, or BothOperators
valid_tokens = {
    "+": BothOperators(UnaryOperator("Positive (+)", op_pos, False), BinaryOperator("Addition (+)", op_add, 4, True)),
    "-": BothOperators(UnaryOperator("Negative (-)", op_neg, False), BinaryOperator("Subtraction (-)", op_sub, 4, True)),
    "*": BinaryOperator("Multiplication (*)", op_mul, 3, True),
    "/": BinaryOperator("Division (/)", op_true_div, 3, True),
    "\\": BinaryOperator("Floor division (\\)", op_floor_div, 3, True),
    "%": BinaryOperator("Mod (%)", op_mod, 3, True),
    "^": BinaryOperator("Exponentiation (^)", op_exp, 2, False),
    "¬": BinaryOperator("Root (¬)", op_root, 2, False),
    "p": BinaryOperator("Permutations (P)", op_permutations, 0, True),
    "c": BinaryOperator("Combinations (C)", op_combinations, 0, True),
    "!": UnaryOperator("Factorial (!)", op_factorial, True),
    "ln": FunctionType("Natural log (ln)", func_ln, 1),
    "log": FunctionType("Logarithm (log)", func_log, 2),
    "abs": FunctionType("Absolute value (abs)", func_abs, 1),
    "lcm": FunctionType("Lowest common multiple", func_lcm, 2),
    "hcf": FunctionType("Highest common factor", func_hcf, 2),
    "rand": FunctionType("Random number generator", func_rand, 2),
    "quadp": FunctionType("Quadratic equation solver (postive square root)", func_quadp, 3),
    "quadn": FunctionType("Quadratic equation solver (negative square root)", func_quadn, 3),
    "sin": FunctionType("Sin (sin)", func_sin, 1),
    "cos": FunctionType("Cosine (cos)", func_cos, 1),
    "tan": FunctionType("Tangent (tan)", func_tan, 1),
    "arsin": FunctionType("Inverse sine (arsin)", func_arsin, 1),
    "arcos": FunctionType("Inverse cosine (arcos)", func_arcos, 1),
    "artan": FunctionType("Inverse tangent (artan)", func_artan, 1),
    "sinh": FunctionType("Hyperbolic sin (sinh)", func_sinh, 1),
    "cosh": FunctionType("Hyperbolic cosine (cosh)", func_cosh, 1),
    "tanh": FunctionType("Hyperbolic tangent (tanh)", func_tanh, 1),
    "arsinh": FunctionType("Inverse hyperbolic sine (arsinh)", func_arsinh, 1),
    "arcosh": FunctionType("Inverse hyperbolic cosine (arcosh)", func_arcosh, 1),
    "artanh": FunctionType("Inverse hyperbolic tangent (artanh)", func_artanh, 1),
    "pi": Num("3.14159265358979323846264338327950288"),
    "tau": Num("6.28318530717958647692528676655900576"),
    "e": Num("2.71828182845904523536028747135266249"),
    "g": Num("9.80665"),
    "phi": Num("1.61803398874989484820458683436563811")
}
```

Operations

I added the code for each operation to ‘Operations.py’:

```
"""
Contains the code for the operations that can be used in the calculator
"""

from Errors import CalcOperationError
from math import log, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh
from random import randint

def op_add(x, y):
    """Return x add y"""

    # typical cases
    return x + y

def op_sub(x, y):
    """Return x subtract y"""

    # typical cases
    return x - y

def op_mul(x, y):
    """Return x multiplied by y"""
```

```
    # typical cases
    return x * y

def op_true_div(x, y):
    """Return x divided by y"""

    # invalid case
    if y == 0:
        raise CalcOperationError("Cannot divide by 0", "/", [x, y])

    # typical cases
    return x / y

def op_floor_div(x, y):
    """Return x divided by y, rounded down"""

    # invalid case
    if y == 0:
        raise CalcOperationError("Cannot divide by 0", "\\", [x, y])

    # typical cases
    return x // y

def op_mod(x, y):
    """Return x mod y - the remainder when x is divided by y"""

    # invalid case
    if y == 0:
        raise CalcOperationError("Cannot divide by 0", "%", [x, y])

    # typical cases
    return x % y

def op_exp(x, y):
    """Return x to the power of y"""

    # invalid case
    if x == 0 and y == 0:
        raise CalcOperationError("0 to the power of 0 is undefined", "^", [x, y])

    # typical cases
    return x ** y

def op_root(root, x):
    """Return the root th root of x"""

    # invalid cases
    if root <= 0 or root % 1 != 0:
        raise CalcOperationError("The root must be a positive whole number", "¬", [root, x])

    # specific case of power
    return x ** (1 / root)

def op_permutations(n, r):
    """Return the number of ways there are to arrange r things in n places, counting all
orders"""

    # invalid cases
    if n % 1 != 0 or r % 1 != 0 or n < 0 or r < 0:
        raise CalcOperationError("Both must be whole numbers and cannot be negative", "P", [n,
r])
    if r > n:
        raise CalcOperationError("r ({}) cannot be greater than n ({})".format(r, n), "P", [n,
r])

    # use the factorial operation already defined
    return op_factorial(n) / op_factorial(n - r)

def op_combinations(n, r):
```

```
    """Return the number of ways there are to arrange r things in n places, only counting 1
order"""

    # invalid cases
    if n % 1 != 0 or r % 1 != 0 or n < 0 or r < 0:
        raise CalcOperationError("Both must be whole numbers and cannot be negative", "C", [n,
r])
    if r > n:
        raise CalcOperationError("r ({}) cannot be greater than n ({})".format(r, n), "C", [n,
r])

    # use the factorial operation already defined
    return op_factorial(n) / (op_factorial(r) * op_factorial(n - r))

def op_pos(x):
    """Return the positive of x"""

    # typical cases
    return +x

def op_neg(x):
    """Return the negative of x"""

    # typical cases
    return -x

def op_factorial(x):
    """Return x factorial"""

    # invalid cases
    if x < 0 or x % 1 != 0:
        raise CalcOperationError("Must be whole number and cannot be negative", "!", [x])

    # multiply all numbers between 1 and x together
    product = 1
    while x > 1:
        product *= x
        x -= 1

    return product

def func_ln(x):
    """Return ln(x)"""

    # invalid cases
    if x <= 0:
        raise CalcOperationError("Can only find the natural log of positive numbers", "ln", [x])

    # use the function from the 'math' library
    return log(x)

def func_log(x, base):
    """Return log of x to the base 'base'"""

    # invalid cases
    if x <= 0 or base <= 0:
        raise CalcOperationError("Can only find the log of a positive number with a positive
base", "log", [base, x])

    # use the function from the 'math' library
    return log(x, base)

def func_abs(x):
    """Return the absolute value of x"""

    # use the built-in function
    return abs(x)

def prime_factors(x):
    """Return a list of all of x's prime factors"""
```

```
    # invalid cases
    if x % 1 != 0 or x < 1:
        raise CalcOperationError("Must be a positive whole number", "LCM or HCF", [x])

    # typical cases
    factors = []
    i = 2

    # for each possible factor
    while i ** 2 <= x:

        # if it is a factor, add it and adjust x to prevent repeats
        # don't increment i as more than 1 of the same factor is possible
        if x % i == 0:
            x //= i
            factors.append(i)

        # if it's not a factor, increment it to find the next
        else:
            i += 1

    # add the last factor if x isn't 1
    if x > 1:
        factors.append(x)

    return factors

def to_dict(factors):
    """Gets a dictionary of all numbers and the number of times they occur"""

    factors_dict = {}
    for factor in factors:

        # if that prime factor is already in the dictionary, increment the count
        if factor in factors_dict:
            factors_dict[factor] += 1

        # otherwise, add it with an initial count of 1
        else:
            factors_dict[factor] = 1

    return factors_dict

def product_dict(dictionary):

    # the number of times each key appears is the value corresponding to that key
    # so the product is the product of all keys each to the power of their value
    product = 1
    for key in dictionary:
        product *= key ** dictionary[key]

    return product

def func_lcm(x, y):
    """Return the lowest common multiple of x and y"""

    # gets dictionaries for each input with the prime factors
    # as the key and the number of occurrences as the value
    prime_factors_x = to_dict(prime_factors(x))
    prime_factors_y = to_dict(prime_factors(y))

    # the prime factors of the LCM of x and y starts as those of x
    prime_factors_lcm = prime_factors_x.copy()

    for factor in prime_factors_y:

        # if the factor is not in LCM or it is in LCM but y has more, add the number y has
        if factor not in prime_factors_lcm or prime_factors_y[factor] >
prime_factors_lcm[factor]:
```

```
            prime_factors_lcm[factor] = prime_factors_y[factor]

    # multiply all the prime factors together to get the LCM
    return product_dict(prime_factors_lcm)

def func_hcf(x, y):
    """Return the highest common factor of x and y"""

    # gets dictionaries for each input with the prime factors
    # as the key and the number of occurrences as the value
    prime_factors_x = to_dict(prime_factors(x))
    prime_factors_y = to_dict(prime_factors(y))

    prime_factors_hcf = {}

    # the prime factors of the HCF are the minimum number that are in both x and y
    for factor in prime_factors_x:
        if factor in prime_factors_y:
            prime_factors_hcf[factor] = min(prime_factors_x[factor], prime_factors_y[factor])

    # multiply all the prime factors together to get the HCF
    return product_dict(prime_factors_hcf)

def func_quadp(a, b, c):
    """Return the positive square root answer of the quadratic equation ax^2 + bx + c = 0"""

    # quadratic formula with positive square root
    return (-b + (b**2 - 4*a*c)**0.5) / (2 * a)

def func_quadn(a, b, c):
    """Return the positive square root answer of the quadratic equation ax^2 + bx + c = 0"""

    # quadratic formula with negative square root
    return (-b - (b**2 - 4*a*c)**0.5) / (2 * a)

def func_rand(low, high):

    # invalid cases
    if low % 1 != 0 or high % 1 != 0:
        raise CalcOperationError("Must be whole numbers", "rand", [low, high])

    # use the function from the 'random' library
    return randint(low, high)

def func_sin(x):
    """Return sin(x) where x is in radians"""

    # use the function from the 'math' library
    return sin(x)

def func_cos(x):
    """Return cos(x) where x is in radians"""

    # use the function from the 'math' library
    return cos(x)

def func_tan(x):
    """Return tan(x) where x is in radians"""

    from Datatypes import valid_tokens, Num

    # invalid cases
    if x % valid_tokens["pi"] == Num("0.5") * valid_tokens["pi"]:
        raise CalcOperationError("Tangent is undefined for values half way between multiples of
pi", "tan", [x])

    # use the function from the 'math' library
    return tan(x)

def func_arsin(x):
```

```
    """Return arsin(x) where the answer is in radians"""

    # invalid cases
    if x < -1 or x > 1:
        raise CalcOperationError("Inverse sine is only defined for values between -1 and 1 
inclusive", "arsin", [x])

    # use the function from the 'math' library
    return asin(x)

def func_arcos(x):
    """Return arcos(x) where the answer is in radians"""

    # invalid cases
    if x < -1 or x > 1:
        raise CalcOperationError("Inverse cosine is only defined for values between -1 and 1 
inclusive", "arcos", [x])

    # use the function from the 'math' library
    return acos(x)

def func_artan(x):
    """Return artan(x) where the answer is in radian"""

    # use the function from the 'math' library
    return atan(x)

def func_sinh(x):
    """Return sinh(x) where x is in radians"""

    # use the function from the 'math' library
    return sinh(x)

def func_cosh(x):
    """Return cosh(x) where x is in radians"""

    # use the function from the 'math' library
    return cosh(x)

def func_tanh(x):
    """Return tanh(x) where x is in radians"""

    # use the function from the 'math' library
    return tanh(x)

def func_arsinh(x):
    """Return arsinh(x) where the answer is in radians"""

    # use the function from the 'math' library
    return asinh(x)

def func_arcosh(x):
    """Return arcosh(x) where the answer is in radians"""

    # invalid cases
    if x < 1:
        raise CalcOperationError("Inverse hyperbolic cosine is undefined for values less than 
1", "arcosh", [x])

    # use the function from the 'math' library
    return acosh(x)

def func_artanh(x):
    """Return artanh(x) where the answer is in radian"""

    # invalid cases
    if x < -1 or x > 1:
        raise CalcOperationError("Inverse hyperbolic tangent is only defined for values between 
-1 and 1 inclusive", "artanh", [x])
```

```
    # use the function from the 'math' library
    return atanh(x)
```

8) Settings

To round, I will create a new subroutine ‘post_calc’ where I would have applied all the other settings but have ran out of time.

At the end of ‘execute’, I converted to a string and converted standard form into my version. I will move these to the new subroutine and round to 15 decimal places using the built-in Python function. However, when some Python numbers are rounded, trailing 0s are kept which isn’t very user friendly so I will remove them.

The following is the end of the ‘execute’ function with the string conversion and standard form replacement removed followed by ‘post_calc’ where they have been moved. This is then followed by the ‘calculate’ method being changed to add the ‘post_calc’ method to it.

```
    # there should be exactly 1 number on the stack at the end - the answer
    # if not, there are too many operands or too few operators
    if len(stack) != 1:
        raise CalcError("Too many operands or too few operators")

    # the last item on the stack is the answer
    return stack.pop()

def post_calc(ans):
    """Apply settings to the answer"""

    # check a valid number
    ans = float(ans)
    if ans == float("inf"):
        raise CalcError("Number too big")

    # round all answers to 15 decimal places
    ans = round(ans, 15)

    # convert to a string so it is in an exact and built-in type
    ans = str(ans)

    # remove trailing 0s
    while ans[-1] == "0":
        ans = ans[:-1]

    # remove the decimal point if it ends in one but nothing after that
    if ans[-1] == ".":
        ans = ans[:-1]

    # if the number is now '-0', make '0'
    if ans == "-0":
        ans = "0"

    # convert back to my representation of standard form
    ans = ans.lower().replace("e", "~")

    return ans
```

```
def calculate(expr, debug=False):
    """
    Calculate the answer to 'expr'.
    If CalcError (or it's child CalcOperationError) has been raised,
    it is due to an invalid expression so needs to be caught and presented as an error message
    Any other exceptions are errors in the code

    :param expr (str): The expression to execute
    :param debug (bool): Whether or not to print out extra information to check for errors. Default: False
    :return ans (str): The answer to 'expr'
    """

    assert isinstance(expr, str), "param 'expr' must be a string"

    for func in [tokenise, convert, execute, post_calc]:

        # execute each function with the result from the last
        expr = func(expr)

        # output the progress if debug is on
        if debug:
            print(str(func).split("function ")[1].split(" at ")[0] + ":", repr(expr))

    return expr
```

The reason I convert to a string (other than to represent standard form in my way) is so it is in a built-in type. I want external programs to be able to use the answers but as I am using a library to represent numbers (‘decimal’) and I have made my own class to wrap it (‘Num’), an external program won’t know what these are but it does know the ‘str’ type as it is built in to Python.

Testing

For each stage, I will write a testing table with all tests I plan to complete. Then as I test each one, I will document all the failed tests, explain the problem and how I fixed them before recording the completed testing table.

When I encounter a failed test, I will fix it before performing the rest of the tests as that error may carry forward and make something else seem as if it is failing when it's the same problem as before.

Once I have fixed all errors, I will retest every test to make sure the fix of a later test hasn't made a previous test fail.

I will test all stages individually using the interfaces stated at the start of each heading below. This means for every error I find, I can be sure that it is from the code I have just written, making it easier to find and fix the error.

1) Core Functionality

Using the command line interface in 'Calc.py':

#	Operation	Type	Description	Expression	Expected Output
1	None	Valid	Integer	1	1
2			Decimal	1.3	1.3
3			'.5' for '0.5' etc	.5	0.5
4			Number surrounded by whitespace	5	5
5		Invalid	Empty expression		Error message
6			Only whitespace		Error message
7			Only brackets	()	Error message
8			Only open bracket	(	Error message
9			Only close bracket	)	Error message
10			Number and open bracket	(1	Error message
11			Number and close bracket	1)	Error message
12	Negative	Valid	Single negative	-1	-1
13			Double negative	--1	1
14			Triple negative	---1	-1
15			Negative with whitespace	- 2	-2
16		Invalid	Only a negative sign	-	Error message
17	Subtraction	Valid	Integers, positive answer	3-1	2
18			Integers, negative answer	1-3	-2
19			Decimals, positive answer	0.3-0.1	0.2
20			Decimals, negative answer	0.1-0.3	-0.2
21			Negative from positive	6.4--3.2	9.6
22			Positive from negative	-6.4-3.2	-9.6
23			Negative from negative	-6.4--3.2	-3.2
24	Exponentiation	Valid	Integers	2^3	8
25			Decimals	0.3^1.6	0.145…
26			Negative base without brackets	-5^2	-25
27			Negative base with brackets	(-5)^2	25
28			Negative base without brackets and a 0	0-5^2	-25
29			Negative power with brackets	2^(-1)	0.5
30			Negative power without brackets	2^-1	0.5
31	Factorial	Valid	Integer	3!	6
32			Negative without brackets	-3!	-6
33		Valid Extreme	0	0!	1
34		Invalid	Negative with brackets	(-1)!	Error message

35			Decimal	1.2!	Error message
36	Many	Valid	Order of operations	1--3!	7
37				4!-3	21
38				2^3!--5	69
39				3!^-2	0.027…
40			Brackets	(2^3)!--5	20325
41			Brackets around whole expression	(2^3!--5)	69
42			Missing close bracket	2-(3!-(-4)	-8
43		Invalid	Missing operand	2^!--5	Error message
44			Missing operator	2^3!5	Error message
45			Missing open bracket	2-(3!)-4)-3	Error message
46	None	Valid	Very big numbers	10^9999999	Error message
47			Call the calculator by importing it from another program	2^8 from import	256

Table 27: Test Plan for 1) Core Functionality

Test 8: Missing close bracket

```
Expr: (
tokenise: [OpenBracket]
convert: Queue(OpenBracket)
Traceback (most recent call last):
  File "C:\Users\James\Documents\Programming\Calculator\CodeBasics\Stage1\Calc.py", line 231, in <module>
    print(calculate(input("Expr: "), True))
  File "C:\Users\James\Documents\Programming\Calculator\CodeBasics\Stage1\Calc.py", line 219, in calculate
    expr = func(expr)
  File "C:\Users\James\Documents\Programming\Calculator\CodeBasics\Stage1\Calc.py", line 197, in execute
    if not token.is_unary:
AttributeError: 'OpenBracket' object has no attribute 'is_unary'
```

This crashed because it thought it was an operator when it was an open bracket. This is actually because open brackets shouldn't be let into 'execute' because one of the main purposes of the shunting yard algorithm is that brackets are not necessary to maintain order of operations in postfix notation.

It got through when I move all remaining items in the stack to the queue at the end. This doesn't happen if there is a matching close bracket because it removes it when the close bracket is parsed. However, when there is not, it stays on the stack until the end.

To fix it, I will check for open brackets when moving everything from the stack to the queue in 'convert':

```
# move everything on the stack to the output queue
while operator_stack:
    token = operator_stack.pop()
    # if an open bracket didn't have a matching closing bracket,
    # it could appear here but we don't want it to be so ignore it
    if not isinstance(token, OpenBracket):
        output_queue.enqueue(token)
```

Reset:

```
>(
tokenise: [OpenBracket]
convert: Queue()
Too many operands or too few operators
```

Test 9: Missing open bracket

```
>)
tokenise: [CloseBracket]
```

A missing open bracket leads to an infinite loop of the program which happens in the close bracket while loop in convert. When it finds a close bracket, it repeatedly loops until it finds an open bracket but if there is no matching open bracket, it loops infinitely. I will change it so if there are no more items in the stack and it hasn't found an open bracket, it errors.

```
# otherwise, it must be a close bracket so add everything from the operator stack to the output queue
# until there is an open bracket, then remove this too but don't add it to the output queue
else:
    while not isinstance(operator_stack.peek(), OpenBracket) and operator_stack.peek() is not None:
        output_queue.enqueue(operator_stack.pop())

    # if there is nothing left in the stack but we haven't
    # found an open bracket, there is a mismatch
    if operator_stack.peek() is None:
        raise CalcError("Too many close brackets or not enough open brackets")

    # remove the open bracket from the stack and discard
    else:
        operator_stack.pop()
```

Retest:

```
>)
tokenise: [CloseBracket]
Too many close brackets or not enough open brackets
```

Test 17: Reversed order of operands

```
Expr: 3-1
tokenise: [Num(3.0), BinaryOperator(subtraction), Num(1.0)]
convert: Queue(Num(3.0), Num(1.0), BinaryOperator(subtraction))
execute: Num(-2.0)
-2.0
```

3-1 = -2? The calculator must be doing 1-3. In execute, I can see that in the stack the operands are in the correct order but because I pop the last one and add this to the list first, I reverse it when adding it to the list. Therefore, I will reverse the list of operands before executing the operation. I will do this with a common python idiom for reversing using slicing '[::-1]'.

```
# execute the operator with its operands (reversed), make it a Num and push it to the stack
stack.push(Num(token.execute(operands[::-1])))
```

Retest:

```
>3-1
tokenise: [Num(3), BinaryOperator(subtraction), Num(1)]
convert: Queue(Num(3), Num(1), BinaryOperator(subtraction))
execute: '2'
2
```

Test 19: Decimal Inaccuracies

```
Expr: 0.3-0.1
tokenise: [Num(0.3), BinaryOperator(subtraction), Num(0.1)]
convert: Queue(Num(0.3), Num(0.1), BinaryOperator(subtraction))
execute: Num(0.19999999999999998)
0.19999999999999998
```

Because $\log_2(10)$ is not an integer, computers cannot store denary numbers exactly in binary – 0.1 is a recurring decimal in binary so they can only store it to a certain accuracy. Despite 15 odd decimal places being quite accurate, I would like to be exact. Python has a library called 'decimal' which maintains accuracy to 28 decimal places of decimal arithmetic. Instead of inheriting my Num class from float, I could inherit from 'Decimal' – a class in the library 'decimal' which will implement all operations float would but to more accuracy.

```
class Num(Decimal):
    """Represents a number"""
    # inherits all methods from Decimal but overrides representation method
    def __repr__(self):
        return "Num({})".format(self)
```

Retest:

```
>0.3-0.1
tokenise: [Num(0.3), BinaryOperator(subtraction), Num(0.1)]
convert: Queue(Num(0.3), Num(0.1), BinaryOperator(subtraction))
execute: '0.2'
0.2
```

Test 30: Power of minus 1

The calculator doesn't seem to be able to handle putting a power of a negative without brackets round it. On further inspection, it does recognise the negative (rather than think it is a subtract) but the postfix is incorrect. It is [2, ^, 1, -] rather than [2, 1, -, ^] so the exponentiation only has 1 operand and it errors. This is because it thinks the exponentiation should be executed first because it has a better precedence than negative.

Some documents show that exponentiation has a higher precedence than unary operators and others the opposite, so I am unsure which to go with. This discussion[4] seems to conclude that there is no standard and brackets are required to decide but what seems mathematically correct follows different rules at different times:

- -2^2 should = -4 as there are no brackets around the -2 so in this case, exponentiation has a higher precedence
- 2^-2 should equal 0.25 as the -2 should be executed first so in this case, negation has a higher precedence

My handheld scientific calculator puts invisible brackets after an exponent sign which solves this problem in its own way. I cannot do this as if I did put brackets after it, the user would not know they were there so it will cause all sorts of problems (this doesn't happen in the calculator as the text inside the brackets is raised up, so it is obvious to the user it is in the brackets). I cannot put visible brackets in the text as they are writing it as I would not know where and with command-line inputs, this is impossible.

A similar problem is when a left associative unary operator comes before an exponentiation too (before it was a *right* associative unary operator coming *after* an exponentiation). For example, 3!^2 works but does 3^2 and then performs factorial on the answer which is clearly wrong:

```
Expr: 3!^2
tokenise: [Num(3), UnaryOperator(factorial), BinaryOperator(exponentiation), Num(2)]
convert: Queue(Num(3), Num(2), BinaryOperator(exponentiation), UnaryOperator(factorial))
execute: Num(362880)
362880
```

The best option seems to be making unary operators have a higher precedence than exponentiation as it only means there are implied brackets around the -2 in -2^4 which is less obviously wrong than the problem above!

I have changed the code to swap the precedences:

```
class UnaryOperator(Operator):
    """
    Represents a unary operator and stores information about it

    :param name (str): The name of the operator
    :param func (identifier): The identifier of the function to execute the operation
    :param is_left_associative (bool): Whether or not the operand is on the left side of the operator (alternative is on the right)
    """

    def __init__(self, name, func, is_left_associative):
        super().__init__(name, func, 1, is_left_associative, True)
```

[4] https://math.stackexchange.com/questions/1299236/why-does-unary-minus-operator-sometimes-take-precedence-over-exponentiation-and

```python
# create the tokens that can be used in the calculator with the classes above and the operations in 'Operations.py'
# the key is the symbol that will be in expressions and the value is an instance of one of the following classes:
# UnaryOperator, BinaryOperator, Operator, or BothOperators
valid_tokens = {
    "-": BothOperators(UnaryOperator("negative", op_neg, False), BinaryOperator("subtraction", op_sub, 4, True)),
    "!": UnaryOperator("factorial", op_factorial, True),
    "^": BinaryOperator("exponentiation", op_exp, 2, False)
}
```

And the new precedence table is:

Precedence	Description
1	Unary operators
2	Exponentiation
3	Multiplication, division, mod, div
4	Addition, subtraction

Table 28: Improved Operator Precedence

Retests:

```
>2^-1
tokenise: [Num(2), BinaryOperator(exponentiation), UnaryOperator(negative), Num(1)]
convert: Queue(Num(2), Num(1), UnaryOperator(negative), BinaryOperator(exponentiation))
execute: '0.5'
0.5

>-3^2
tokenise: [UnaryOperator(negative), Num(3), BinaryOperator(exponentiation), Num(2)]
convert: Queue(Num(3), UnaryOperator(negative), Num(2), BinaryOperator(exponentiation))
execute: '9'
9
```

Test 46: Very large numbers

```
>10^9999999
Traceback (most recent call last):
  File "Calc.py", line 351, in <module>
    print(calculate(expression))
  File "Calc.py", line 334, in calculate
    expr = func(expr)
  File "Calc.py", line 273, in execute
    token = token.execute(operands[::-1])
  File "C:\Users\james\Dropbox\Programming\Calculator\CodeBasics\8-Settings\Tested\Datatypes.py", line 133, in execute
    return self.func(*operands)
  File "C:\Users\james\Dropbox\Programming\Calculator\CodeBasics\8-Settings\Tested\Operations.py", line 65, in op_exp
    return x ** y
decimal.Overflow: [<class 'decimal.Overflow'>]
```

This led to an overflow error specific to the ‘decimal’ library which I need to catch.

In ‘execute’, I will catch many of the ‘decimal’ library’s exceptions and make them my own that will then be presented to the user as an error message:

```
# execute the operator with its operands (reversed)
# catch errors raised by the 'decimal' library and convert them to my format
try:
    token = token.execute(operands[::-1])
except InvalidOperation:
    raise CalcError("Invalid operation")
except Overflow:
    raise CalcError("Number too big")
except DecimalException as e:
    raise CalcError("Error: " + str(e).split("decimal.")[1].split("'>]")[0])

# make it a 'Num' and push it to the stack
stack.push(Num(token))
```

Retest:

```
>10^9999999
Number too big
```

Test 47: Importing from another program

I only need to test this once to check it works as the specific things I can do with the calculator are the same as before, so my other tests check they work.

I will create a file that imports it, calls it with an expression and prints the answer:

```
from Calc import calculate
print(calculate("6*7"))
```

Output from running that file:

```
C:\Users\james\Dropbox\Programming\Calculator\CodeBasics\1-Core\Tested>python EXT.py
256
```

Testing Table

#	Actual Output	PASS/FAIL	Comment	Retest Output	Retest PASS/FAIL
1	1.0	PASS	Becomes a decimal but will be rounded in a later stage		
2	1.3	PASS			
3	0.5	PASS			
4	5.0	PASS			
5	Error message	PASS			
6	Error message	PASS			
7	Error message	PASS			
8	Crash	FAIL	It crashed because an open bracket ended up in the queue where it's not meant to be which I need to fix	Error message	PASS
9	Infinite loop	FAIL	Caused an infinite loop which needs to be fixed	Error message	PASS

10	1.0	PASS	Implied close bracket at the end of the expression which works correctly despite me expecting it to fail		
11	Error message	PASS			
12	-1.0	PASS			
13	1.0	PASS			
14	-1.0	PASS			
15	-2.0	PASS			
16	Error message	PASS			
17	-2.0	FAIL	Opposite of what I want – the order of the operands must be reversed	2.0	PASS
18	-2.0	PASS			
19	0.199…998	FAIL	Should be exactly 0.2 but computers can't store denary numbers in binary accurately	0.2	PASS
20	-0.2	PASS			
21	9.6	PASS			
22	-9.6	PASS			
23	-3.2	PASS			
24	8	PASS			
25	0.145…	PASS			
26	-25	PASS	Needed a retest after the change of code after test 30	25	PASS
27	25	PASS			
28	-25	PASS	Needed a retest after the change of code after test 30	-25	PASS
29	0.5	PASS			
30	Error message	FAIL	Should be able to deal with that so changed operator precedence. This needs a retest of tests 26 and 28	0.5	PASS
31	6	PASS			
32	-6	PASS	Executes the factorial first and applies the negative to the answer so this is correct		
33	1	PASS			
34	Error message	PASS			
35	Error message	PASS			
36	7	PASS			
37	21	PASS			
38	69	PASS			
39	0.027…	PASS			
40	40325	PASS			
41	69	PASS			
42	-8	PASS	Implied close bracket at end of expression		
43	Error message	PASS			
44	Error message	PASS			
45	Error message	PASS			

46	Crash	FAIL	Overflow error which I need to catch	Error message	PASS
47	256	PASS			

Table 29: Testing Table for 1) Core Functionality

2) Interface (Memory)

Using the command line interface in 'Interface.py':

#	Type	Description	Expression	Expected Output
1	Invalid	Display memory when it is empty	memory	Error message
2	Valid	Use the calculator	3-1	2
3			2^3	8
4		Display the memory when there are items in memory	memory	1: 2^3 = 8 2: 3-1 = 2
5		Recall a memory item	M2	2
6		Clear the memory	clear	Memory cleared
7	Invalid	Check the memory has been cleared by attempting to display it	memory	Error message
8		Reference most recent memory when it is empty	ans	Error message
9		Reference memory when empty	M2	Error message
10	Valid	Display the instructions	instructions	Instructions
11		Clear memory when empty	clear	Memory cleared
12		Use the calculator	3!	6
13	Invalid	Reference memory that doesn't exist, but memory is not empty	M2	Error message
14	Invalid Extreme	Reference memory 0	M0	Error message
15	Valid	Reference most recent memory	ans	6
16		Use most recent answer in a calculation	ans^2	36
17		Use a memory reference in a calculation	M2^2	36
18		Using an upper case ANS	ANS	36
19		Using multiple references in the same expression	ans-M2	0

Table 30: Test Plan for 2) Interface (Memory)

Test 1: Empty Memory

```
>memory

>
```

It doesn't error but it should show an error message, or it just looks as if nothing has happened. I will add a check for this with the else block after the for loop:

```
# typing 'recent memory' will output all previous calculations
elif "memory" in expression:
    for expr, ans in calc.recent_memory():
        print("{} = {}".format(expr, ans))
    else:
        print("Memory is empty")
```

Retest:

```
>memory
Memory is empty
```

Test 4: Numbering Memory Items and still saying empty

Although not erroring, the memory items need to be numbered according to the x value they would have to use if they wanted to reference it by typing ‘Mx’. To do this, when outputting them I will keep track of a ‘count’ variable and output this too:

```
# typing 'recent memory' will output all previous calculations
elif "recent memory" in expression:
    count = 1
    for expr, ans in calc.recent_memory():
        print("{}: {} = {}".format(count, expr, ans))
        count += 1
```

I had also misunderstood the ‘else’ block after a ‘for’ block. I thought it would only run when the for loop doesn’t run but it actually runs whenever the ‘break’ keyword isn’t used in the for loop. This is why it was running every time it is called as I don’t use the ‘break’ keyword.

I can fix this using the code I added to number them. If it hasn’t displayed any memory items, ‘count’ will still be 1 so I can check for this afterwards:

```
# typing 'memory' will output all previous calculations
elif "memory" in expression:
    count = 1
    for expr, ans in calc.recent_memory():
        print("{}: {} = {}".format(count, expr, ans))
        count += 1
    if count == 1:
        print("Memory is empty")
```

Retest:

```
>memory
1: 2^3 = 8
2: 3-1 = 2
```

Test 8: Checking if empty before using ‘ans’

```
>ans
Traceback (most recent call last):
  File "Interface.py", line 149, in <module>
    expression = expression.replace("ans", calc.memory_item()[
1])
  File "Interface.py", line 82, in memory_item
    raise IndexError("There aren't that many items saved in me
mory")
IndexError: There aren't that many items saved in memory
```

When the user types ‘ans’, I don’t check whether or not memory is empty so if it is, it errors. I need to check for this:

```
# including 'ans' will replace it with the previous answer
if "ans" in expression:
    if calc.len_memory() == 0:
        raise CalcError("Memory is empty")
    else:
        expression = expression.replace("ans", calc.memory_item()[1])
```

I do check for 1 error and raise another but the error I raise is caught below anyway and the message is displayed. This means the code below doesn’t execute to not cause another error.

Retest:

```
>ans
Memory is empty
```

Test 17: Reference in Calculation

```
>M2^2
Memory references must be a whole numbers
```

When I reference memory using ‘Mx’ within a calculation, the way I find x is by taking whatever is between the ‘M’ and the next space. This is why ‘M2^2’ doesn’t work but ‘M2 ^2’ would. This is not intuitive, so I need to fix it. I need to stop scanning when the next character is not a digit:

```
# including 'Mx' will replace it with the xth previous answer
while "M" in expression:

    # take all digits between the 'M' and the first non-digit
    index = expression.split("M")[1]
    pos = 0
    while pos < len(index) and index[pos] in "0123456789":
        pos += 1
    index = index[:pos]

    if index == "":
        raise CalcError("You must give a memory reference after 'M'")

    try:
        index = int(index)
    except ValueError:
        raise CalcError("Memory references must be whole numbers")
    else:
        if index > calc.len_memory():
            raise CalcError("Not enough items in memory")
        elif index < 1:
            raise CalcError("Memory references must be greater than or equal to 1")
    expression = expression.replace("M{}".format(index), calc.memory_item(index)[1], 1)
```

The slice gets all characters up to ‘pos’ positions through the string. The string is already only characters after ‘M’.

Retest:

```
>M2^2
36
```

Test 18: Case

```
>ANS
Invalid token: 'a'
```

The case matters of ‘M’ and ‘ans’ when it shouldn’t. Because of this, I will lower the string before I parse it and check for the lower-case versions:

```
expression = input("\n>").lower()
```

Retest:

```
>ANS
36
```

Testing Table

#	Actual Output	PASS/FAIL	Comment	Retest Output	Retest PASS/FAIL
1	Nothing	FAIL	It should display something so the user knows it worked	Error message	PASS
2	2	PASS			

3	8	PASS			
4	2^3 = 8 3-1 = 2 Memory is empty	FAIL	They should be numbered and it shouldn't say memory is empty	1: 2^3 = 8 2: 3-1 = 2	PASS
5	2	PASS			
6	Memory cleared	PASS			
7	Error message	PASS			
8	Crash	FAIL	Doesn't check whether memory is empty before using ans	Error message	PASS
9	Error message	PASS			
10	Instructions	PASS			
11	Memory cleared	PASS			
12	6	PASS			
13	Error message	PASS			
14	Error message	PASS			
15	6	PASS			
16	36	PASS			
17	Error message	FAIL	Scanning the text wrong	36	PASS
18	Error message	FAIL	Doesn't lower the text in 'Interface.py'	36	PASS
19		PASS			

Table 31: Testing Table for 2) Interface (Memory)

3) Graphical User Interface

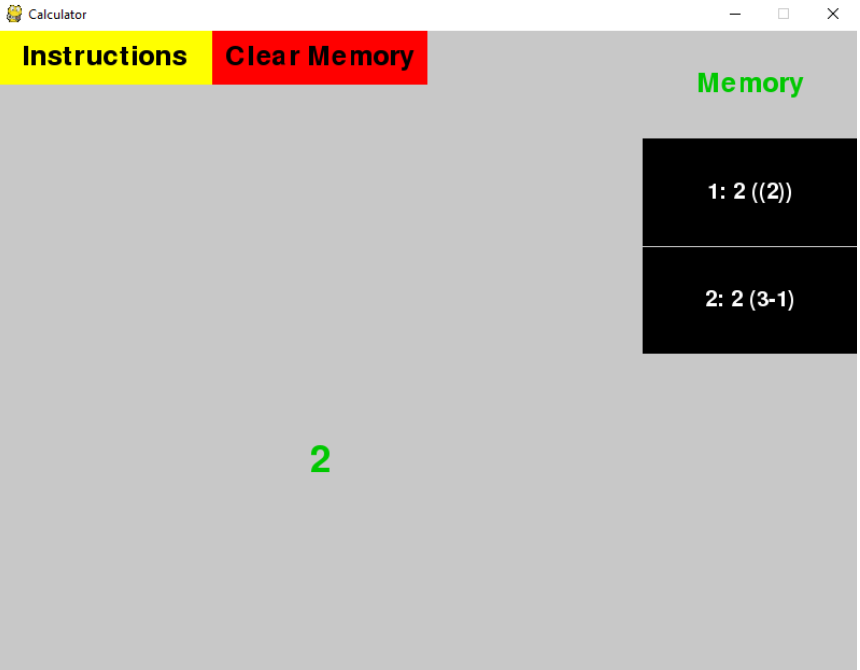
Using the graphical user interface in 'UserInterface.pyw':

#	Type	Description	Inputs	Expected Output
1	Valid	Use the calculator as normal and view memory	Type '3-1' and press RETURN	'2' in the answer area and a memory box containing '2' and '3-1'
2		Recall memory	Click on the memory box and press RETURN	'M1' appears in the expression area and then '2' appears in the answer area and another memory box appears containing '2' and '2'
3		View instructions	Click the instructions button	Go to instructions view
4		View all the instructions	Scroll down	The page scrolls to reveal more instructions
5		View the first instructions again	Scroll up	The page scrolls to reveal the first instructions again
6		Go back to normal mode	Click the back button	Go to normal view
7		Use a previous answer in an expression	Type '2^' then click on the most recent memory box and press RETURN	'2^M1' appears in the expression area and then '4' appears in the answer area and another memory box appears containing '4' and '2^2'
8		Clear the memory	Click the clear memory button	Memory items disappear
9	Invalid	Reference non-existent memory	Type 'ans' and press RETURN	An error message appears in the error message area
10		Give an invalid memory reference	Type 'M-2'	An error message appears in the error message area
11		Give an invalid memory reference	Type 'M1.2'	An error message appears in the error message area

12	Valid	Overflow the expression box	Repeatedly type '9' until a '...' appears at the end of the expression area then press RETURN	5 lines of '9's ending in a '...' in the expression area then the same in the answer area and most recent memory box
13		Referencing memory with 'ans'	Type 'ans' and press RETURN	The same sequence of '9's as before and another memory box as the same
14	Invalid	Give an invalid memory reference	Type 'M1.2' and press RETURN	An error message appears in the error message area
15	Valid	Exit	Click the red cross in the top right corner	Window closes

Table 32: Test Plan for 3) Graphical User Interface

Test 2: Extra Brackets



Because I add brackets to the memory answers before adding them to the expression so it doesn't mess up the expression, every time you reference memory, extra brackets will be added in. To fix this, when saving the expression to memory, I will remove extra brackets on the outside of the expression.

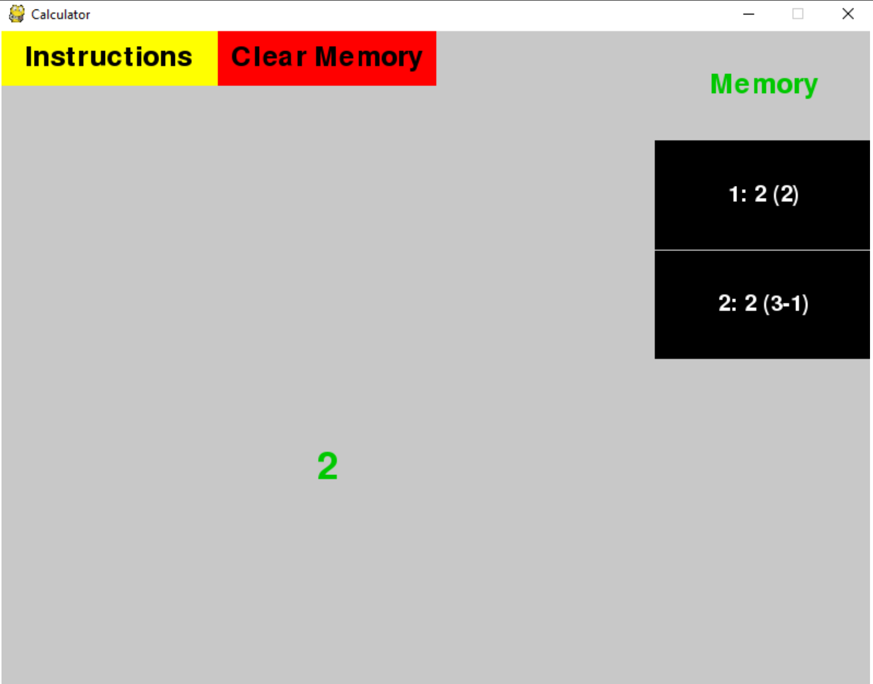
I will write an extra method for the 'Window' class which will remove whitespace and brackets from the outside of the expression:

```
def __clean_up_expr(self, expr):
    """Remove whitespace and extra brackets on the outside of the expression"""

    expr = expr.strip()
    while expr[0] == "(" and expr[-1] == ")":
        expr = expr[1:-1].strip()

    return expr
```

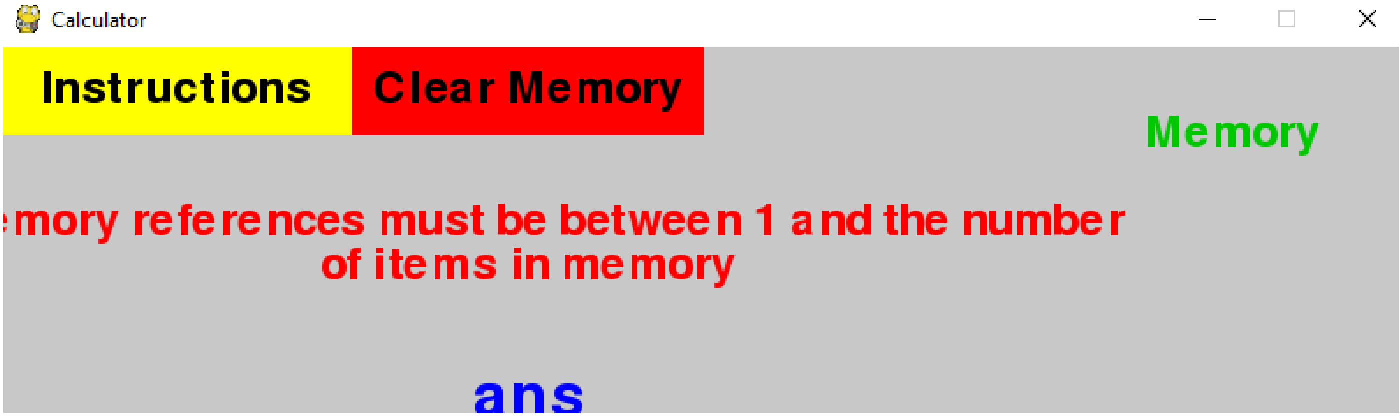
Then I just added it to the line to calculate the answer:

```
self.__ans = self.__calculator.calculate(self.__clean_up_expr(self.__convert_memory_references(expr)))
```

Retest:



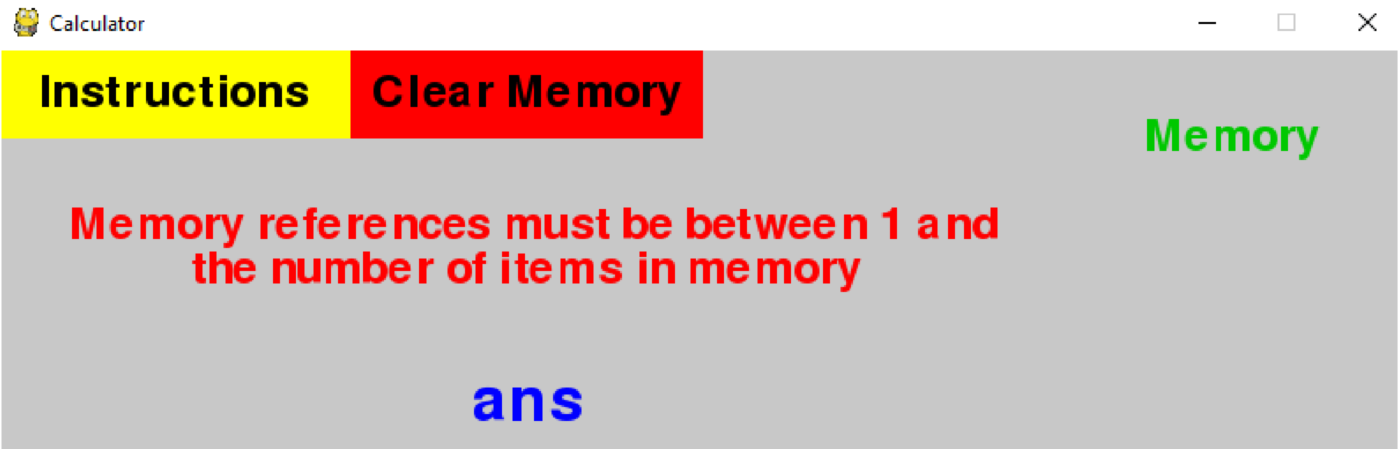
Test 9: Error Message Overflowing Screen



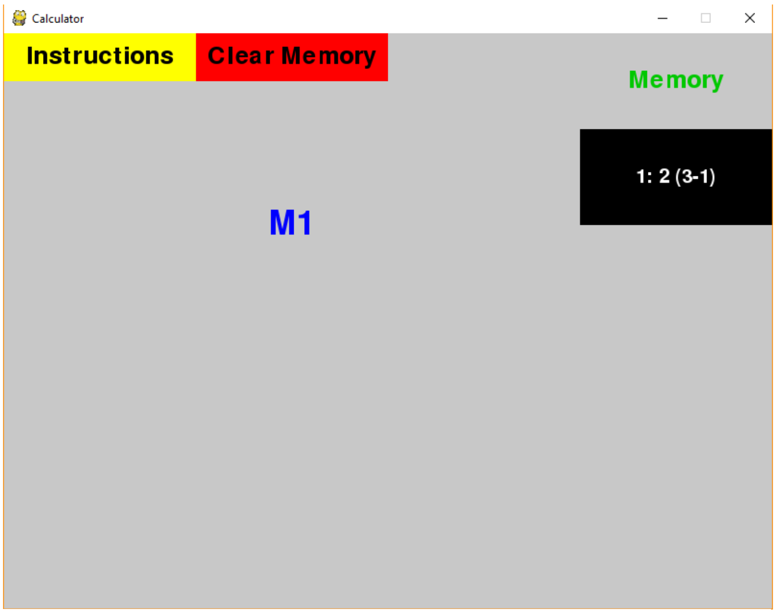
The error message overflows the screen, so I need to reduce the number of characters I allow on 1 line from 50 to 40:

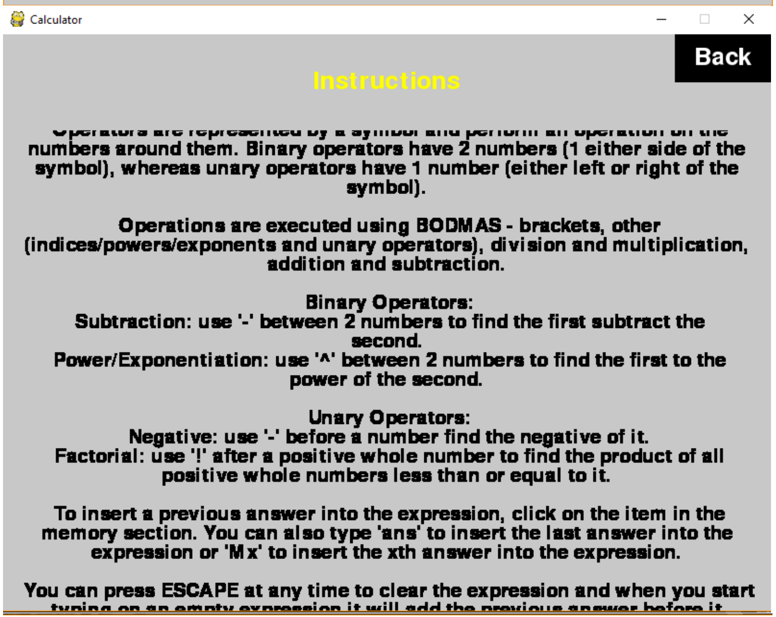
```
lines = format_text(self.__error_msg, 40, 2)
```

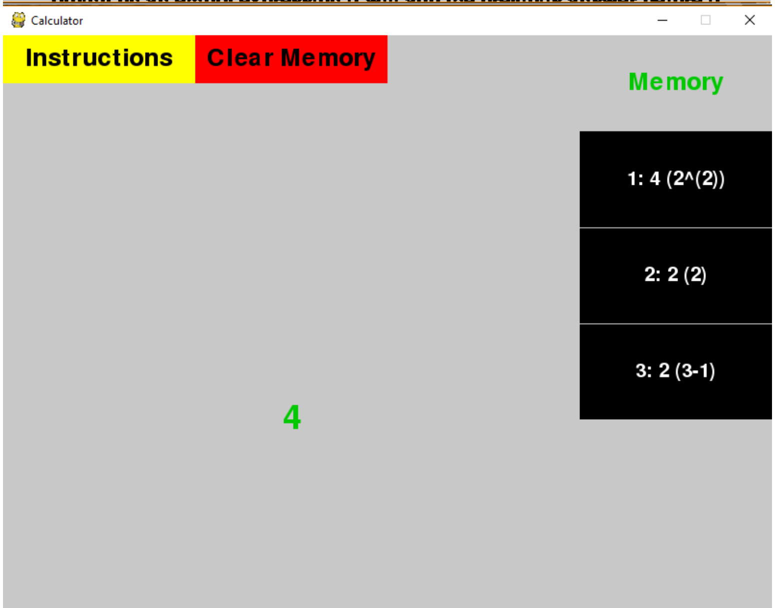
Retest:



Some Successful Tests







Testing Table

#	Actual Output	PASS/FAIL	Comment	Retest Output	Retest PASS/FAIL
1	'2' in the answer area and a memory box showing '1: 2 (3-1)'	PASS			
2	'M1' appears in the expression area and then '2' appears in the answer area and another memory box appears showing '1: 2 ((2))'	FAIL	Although not major, the extra pair of brackets will stack up so I will remove them	'M1' appears in the expression area and then '2' appears in the answer area and another memory box appears showing '1: 2 (2)'	PASS
3	Changed to instructions view	PASS			
4	Scrolled down and stopped when everything was visible	PASS			
5	Scrolled back up and stopped when at the top again	PASS			
6	Changed back to normal view with all memory still there	PASS			
7	'2^M1' appeared in the expression area and then '4' appears in the answer area and another memory box appears showing '1: 4 (2^(2))'	PASS	Can't do anything about brackets within the expression		
8	All memory items and the answer area cleared	PASS			
9	Error message in error area: 'Memory references must be between 1 and the number of items in memory' however this overflows the screen	FAIL	I need to adjust the number of characters that fit on the line	Error message in error area: 'Memory references must be between 1 and the number of items in memory' without overflowing the screen	PASS
10	Error message in error area: 'Invalid token: 'm''	PASS	Although this is a strange error message as it is not related to memory, it is correct as it has been passed to the calculator		
11	Error message in error area: 'Memory references must be between 1 and the number of items in memory'	PASS			
12	5 lines of '9's ending in a '…' in the expression	PASS			

	area and then the same in the answer area and most recent memory box				
13	The same sequence of '9's as before and another memory box as the same	PASS			
14	Error message in error area: 'Too many operands or too few operators'	PASS	Although another strange error message as not related to memory, it has replaced 'M1' with the memory item and then passed that with a '.2' to the calculator which is undefined		
15	Window closes	PASS			

Table 33: Testing Table for 3) Graphical User Interface

4) Constants

Using the command line interface in 'Calc.py':

#	Type	Description	Expression	Expected Output
1	Valid	Constant	e	2.718…
2		Constant with whitespace	e	2.718…
3		Operations on constants	-e	-2.718…
4			pi-e	0.423…
5			e^pi	23.140…
6		Operations on constants with whitespace	e ^ pi	23.140…
7	Invalid	Unknown constant	tau	Error message

Table 34: Test Plan for 4) Constants

Some Successful Tests

```
>e
tokenise: [Num(2.718281828459045235360287471350)]
convert: Queue(Num(2.718281828459045235360287471350))
execute: '2.718281828459045235360287471350'
2.718281828459045235360287471350

>e ^ pi
tokenise: [Num(2.718281828459045235360287471350), BinaryOperato
r(exponentiation), Num(3.141592653589793238462643383270)]
convert: Queue(Num(2.718281828459045235360287471350), Num(3.141
592653589793238462643383270), BinaryOperator(exponentiation))
execute: '23.14069263277926900572908637'
23.14069263277926900572908637

>tau
Invalid token: 'tau'
```

Testing Table

#	Actual Output	PASS/FAIL	Comment	Retest Output	Retest PASS/FAIL
1	2.718…	PASS			
2	2.718…	PASS			
3	-2.718…	PASS			

4	0.423…	PASS			
5	23.140…	PASS			
6	23.140…	PASS			
7	Error message	PASS			

Table 35: Testing Table for 4) Constants

5) Standard Form

Using the command line interface in 'Calc.py':

#	Type	Description	Expression	Expected Output
1	Valid	Integer, positive exponent	24~+5	2400000
2		Decimal, positive exponent	1.2~+5	120000
3		Integer, negative exponent	1~-1	0.1
4		Decimal, negative exponent	3.54~-2	0.0354
5		Unsigned exponent	4.2~2	420
6		Output in standard form	0.0000000000000000000000000000001	1~-31
7		Used in expression	6.67~-2-2.4~2	-239.9333
8		Used in expression	1.7~2^2	28900
9		Exponent of 0	10~0	10
10		Number of 0	0~2	0
11	Invalid	Invalid standard form notation	e~10	Error message
12		Invalid standard form notation	1.6~pi	Error message
13		E not ~	1.2e-2	Error message
14	Valid	Very big numbers	1~999	Error message
15		Very small numbers	1~-999	0

Table 36: Test Plan for 5) Standard Form

Some Successful Tests

```
>24~+5
tokenise: [Num(2.4E+6)]
convert: Queue(Num(2.4E+6))
execute: '2.4~+6'
2.4~+6

>0.0000000000000000000000000000001
tokenise: [Num(1E-31)]
convert: Queue(Num(1E-31))
execute: '1~-31'
1~-31

>6.67~-2-2.4~2
tokenise: [Num(0.0667), BinaryOperator(subtraction), Num(2.4E+
2)]
convert: Queue(Num(0.0667), Num(2.4E+2), BinaryOperator(subtra
ction))
execute: '-239.9333'
-239.9333

>1.6~pi
Invalid token: '~'
```

Testing Table

#	Actual Output	PASS/FAIL	Comment	Retest Output	Retest PASS/FAIL

1	2.4~+6	PASS	It normalised by only allowing 1 non-zero digit before the '~'		
2	1.2~+5	PASS	It's not displaying it in normal form but it's still the same		
3	0.1	PASS			
4	0.0354	PASS			
5	4.2~+2	PASS			
6	1~-31	PASS			
7	-239.9333	PASS			
8	2.89~+4	PASS			
9	10	PASS			
10	0~+2	PASS	This is equal to 0, the 'decimal' library just maintains significance so as I presented it with an exponent of 2, it keeps it that way		
11	Invalid token '~'	PASS			
12	Invalid token '~'	PASS			
13	Too many operands or too few operators	PASS	Implied multiplication hasn't been added yet		
14	Error message	PASS			
15	0	PASS			

Table 37: Testing Table for 5) Standard Form

6) Functions

Using the command line interface in 'Calc.py':

#	Type	Description	Expression	Expected Output
1	Valid	Single function call with 1 argument	sin(pi)	0
2		Single function call with 2 arguments	root(2,9)	3
3		Single function call with 3 arguments	sum(1,2,3)	6
4		Single function call with whitespace around operands	sum(1 , 2 , 3)	6
5		Single function call with whitespace around brackets	sin (pi)	0
6		Operations on functions	sin(pi)-root(2,9)	-3
7		Nested functions	sum(sin(pi),root(2,9),5)	8
8		Many nested functions	sin(sin(sin(sin(sin(sin(sin(sin(sin(sin(pi))))))))))	0
9		Missing close bracket	sin(pi	0
10	Invalid	Missing open bracket	sin pi)	Error message
11		Missing open and close brackets	sin pi	Error message
12		Invalid function	si(pi)	Error message
13		No operands with brackets	sin()	Error message
14		Not enough operands	root(2)	Error message

15		Too many operands	sum(1,2,3,4)	Error message
16		0th root of something	root(0, 1)	Error message

Table 38: Test Plan for 6) Functions

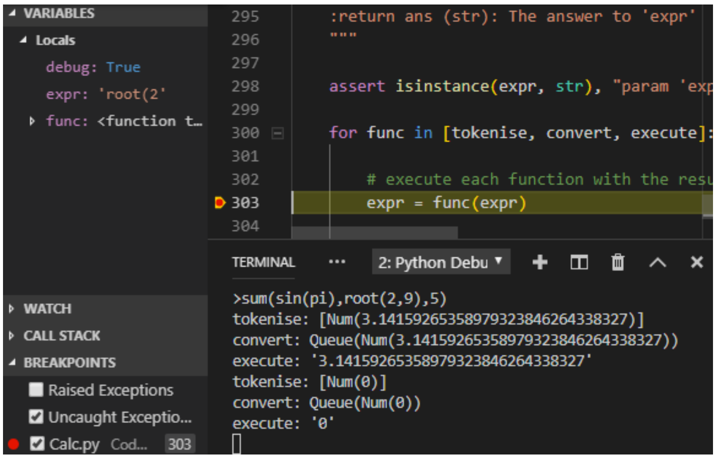
Test 7: Nested Functions

The second function failed in the following nested function call:

```
>sum(sin(pi),root(2,9),5)
tokenise: [Num(3.14159265358979323846264338327)]
convert: Queue(Num(3.14159265358979323846264338327))
execute: '3.14159265358979323846264338327'
tokenise: [Num(0)]
convert: Queue(Num(0))
execute: '0'
tokenise: [root()]
Too many close brackets or not enough open brackets
```

'sin(pi)' executed correctly to be '0' but when it got to root, it errored.

Upon further inspection by debugging, it became clear that the comma in the inner function 'root(2,9)' was being read as if it was for the outer function, meaning the operands for the outer function 'sum' were 'sin(pi)', 'root(2', '9)', and '5'. In this way, it splits any nested function with more than 1 parameter up.



To fix this, in 'tokenise' when I scan the expression for commas and brackets when inside the string, I need to check whether I am in a nested function and if so, don't treat commas as the end of operands. I can use 'bracket_depth' to do this as for there to be a nested function, there has to be a bracket pair and if there is only a bracket pair, it still works as you can't have a bracket pair without a function with a comma inside, e.g.: 'sum((3,2),4)'. This is invalid as the brackets make '(3,2)' the first argument (although invalid) and '4' the second argument with no third argument.

When I check whether there is a comma, I can only do that if 'bracket_depth' is 1 as 0 means it is outside the function's brackets and all commas have to be inside those but no others to count for that function.

```
# if it's a comma, add the operand to the function
# and update the start pos for the next operand
elif key == "comma" and bracket_depth == 1:
    identify_operand(expr[operand_start_pos:pos], tokens[-1])
    operand_start_pos = pos - 1
```

Retest:

```
>sum(sin(pi),root(2,9),5)
tokenise: [Num(3.141592653589793238462643383279502884)]
convert: Queue(Num(3.141592653589793238462643383279502884))
execute: '3.141592653589793238462643383279502884'
tokenise: [Num(0)]
convert: Queue(Num(0))
execute: '0'
tokenise: [Num(2)]
convert: Queue(Num(2))
execute: '2'
tokenise: [Num(9)]
convert: Queue(Num(9))
execute: '9'
tokenise: [Num(3.000000000000000000000000000)]
convert: Queue(Num(3.000000000000000000000000000))
execute: '3.000000000000000000000000000'
tokenise: [Num(5)]
convert: Queue(Num(5))
execute: '5'
tokenise: [Num(8.000000000000000000000000000)]
convert: Queue(Num(8.000000000000000000000000000))
execute: '8.000000000000000000000000000'
8.000000000000000000000000000
```

Test 9: Implied Close Brackets

```
>sin(pi
tokenise: [sin()]
Too many close brackets or not enough open brackets
```

The calculator should be able to deal with omitted brackets around functions – it should add them until all function brackets are closed. I can do this by adding a close bracket if the current position through the expression is overflowing the end so before it checks the while condition and quits, it adds a character, meeting the while condition so it carries on. This can happen for however many brackets are missing.

```
# add omitted closing brackets around functions
elif pos >= len(expr):
    expr += ")"
```

Retest:

```
>sin(pi
tokenise: [Num(3.141592653589793238462643383279502884)]
convert: Queue(Num(3.141592653589793238462643383279502884))
execute: '3.141592653589793238462643383279502884'
tokenise: [Num(0)]
convert: Queue(Num(0))
execute: '0'
0
```

Test 14: Wrong number of operands

```
>root(2)
tokenise: [Num(2)]
convert: Queue(Num(2))
execute: '2'
Traceback (most recent call last):
  File "Calc.py", line 320, in <module>
    print(calculate(expression, True))
  File "Calc.py", line 303, in calculate
    expr = func(expr)
  File "Calc.py", line 169, in tokenise
    tokens[-1] = tokens[-1].execute()
  File "C:\Users\1892-FAMILY\Dropbox\Programming\Calculator\CodeBas
ics\6-Functions\Untested\Datatypes.py", line 239, in execute
    return Num(self.__func(*self.__operands))
TypeError: func_root() missing 1 required positional argument: 'y'
```

If a function is called with the wrong number of operands, it crashed the program with a python error. To fix this, I need to check that there are the correct number of operands before passing them to the function and for this, I need to store the correct number of operands in the function objects and check that there are the correct number before executing:

```python
class FunctionType:
    """
    Represents a type of function and stores information about it

    :param name (str): The name of the type of function
    :param func (identifier): The identifier of the function to execute the operation
    :param num_operands (int): The number of operands the function takes
    """

    def __init__(self, name, func, num_operands):
        self.__name = name
        self.__func = func
        self.__num_operands = num_operands

    def create(self, calc):
        """
        Return a new object which has the same properties as this object
        but is unique for all instances of the function in the expression

        :param calc (function): The calculate function from the main calculator
        :return (object): An instance of the 'FunctionInstance' class
        """

        return FunctionInstance(self.__name, self.__func, self.__num_operands, calc)

    def __repr__(self):
        return "FunctionType({})".format(self.__name)
```

```python
class FunctionInstance:
    """
    Represents a function instance and stores information about it

    :param name (str): The name of the type of function
    :param func (function): The function to execute the operation
    :param num_operands (int): The number of operands the function takes
    :param calc (function): The calculate function from the main calculator
    """

    def __init__(self, name, func, num_operands, calc):
        self.__name = name
        self.__func = func
        self.__num_operands = num_operands
        self.__calc = calc
        self.__operands = []

    def add_operand(self, operand):
        """
        Execute the operand with the calculator to simplify it and then add it to the stored operands

        :param operand (num): The operand to add
        """

        self.__operands.append(Num(self.__calc(operand)))

    def execute(self):
        """
        Recursively execute all operands using the calculator and then
        return the answer when the function is executed with its operands

        :return (Num): The answer to the function when executed on its operands
        """

        if len(self.__operands) != self.__num_operands:
            raise CalcError("{} operands required in {} function call".format(self.__num_operands, self.__name))

        # execute the function with its operands and return it
        # the star splits the list out into individual arguments
        return Num(self.__func(*self.__operands))

    def __repr__(self):
        return "{}({})".format(self.__name, ", ".join([str(operand) for operand in self.__operands]))
```

```python
# create the tokens that can be used in the calculator with the classes above and the operations in 'Operations.py'
# the key is the symbol that will be in expressions and the value is an instance of one of the following classes:
# UnaryOperator, BinaryOperator, Operator, or BothOperators
valid_tokens = {
    "-": BothOperators(UnaryOperator("negative", op_neg, False), BinaryOperator("subtraction", op_sub, 4, True)),
    "!": UnaryOperator("factorial", op_factorial, True),
    "^": BinaryOperator("exponentiation", op_exp, 2, False),
    "pi": Num("3.14159265358979323846264338327"),
    "e": Num("2.71828182845904523536028747135"),
    "sin": FunctionType("sin", func_sin, 1),
    "root": FunctionType("root", func_root, 2),
    "sum": FunctionType("sum", func_sum, 3)
}
```

Retest:

```
>root(2)
tokenise: [Num(2)]
convert: Queue(Num(2))
execute: '2'
2 operands required in root function call
```

Testing Table

#	Actual Output	PASS/FAIL	Comment	Retest Output	Retest PASS/FAIL
1	0	PASS			
2	3.0000000…	PASS			
3	6	PASS			
4	6	PASS			
5	0	PASS			
6	-3.000000…	PASS			
7	Too many close brackets or not enough open brackets	FAIL	Perfectly valid function so need to fix	8.0000000…	PASS
8	0	PASS			
9	Too many close brackets or not enough open brackets	FAIL	It should be able to imply close brackets at the end of the expression	0	PASS
10	Too many close brackets or not enough open brackets	PASS			
11	Too many close brackets or not enough open brackets	PASS			
12	Invalid token 'si'	PASS			
13	Too many operands or too few operators	PASS			
14	Crash	FAIL	This should not crash but just display an error message	2 operands required in sum function call	PASS
15	3 operands required in sum function call	PASS			
16	While performing root with 0, 1: Cannot find the 0th root	PASS			

Table 39: Testing Table for 6) Functions

7) Operations

Using the command line interface in ‘Calc.py’:

#	Operation	Type	Description	Expression	Expected Output
1	Addition	Valid	Integers	1+1	2
2			Decimals	1.5 + 2.3	3.8
3		Invalid	No second operand	6+	Error message
4	Positive	Valid	Integer	+1	1
5			Decimal	+3.2	3.2
6			Chain	++2.98	2.98
7	Subtraction	Valid	Integers	2-1	1
8			Decimals	3.6-1.9	1.7
9		Invalid	No second operand	2-	Error message
10	Negative	Valid	Integer	-1	-1
11			Decimal	-2.3	-2.3
12	Multiplication	Valid	Integers	2*3	6

13			Decimals	1.5*0.3	0.45
14			Zero	0*2	0
15	Division	Valid	Integer answer	6/3	2
16			Decimal answer	3/2	1.5
17		Invalid	Division by 0	2/0	Error message
18	Floor division	Valid	Divisible	6\3	2
19			Not divisible	3\2	1
20		Invalid	Division by 0	2\0	Error message
21	Mod	Valid	Divisible	6%2	0
22		Valid	Not divisible	6%4	2
23		Invalid	Division by 0	6%0	Error message
24	Exponentiation	Valid	Positive exponent	2^3	8
25			Negative exponent	2^-1	0.5
26			Decimals	1.2^2.6	1.60…
27		Invalid	0^0	0^0	Error message
28	Root	Valid	Integers	2¬2	1.41…
29			Decimal	2¬1.5	1.22…
30		Invalid	Negative root	-0.2¬2	Error message
31			Zero root	0¬2	Error message
32	Permutations	Valid	Integers	3P2	6
33		Valid Extreme	Zero r	3P0	1
34			Zero n	0P0	1
35		Invalid	Decimals	1.2P2.2	Error message
36			Negative n	-2P1	Error message
37			Negative root	2P-1	Error message
38			r > n	1P2	Error message
39	Combinations	Valid	Integers	3C2	3
40		Valid Extreme	Zero r	3C0	1
41			Zero n	0C0	1
42		Invalid	Decimals	1.2C2.2	Error message
43			Negative n	-2C1	Error message
44			Negative root	2C-1	Error message
45	Factorial	Valid	Positive integer	3!	6
46		Valid Extreme	Zero	0!	1
47		Invalid	Decimal	1.2!	Error message
48			Negative	-3!	Error message
49	Natural log	Valid	Integer	ln(2)	0.693…
50			Decimal	ln(1.2)	0.182…
51		Invalid	0	ln(0)	Error message
52			Negative	ln(-1)	Error message
53	Log	Valid	Integers	log(8,2)	3
54			Decimal answer	log(1.5,2)	0.584…
55		Invalid	Zero answer	log(0,1)	Error message
56			Zero base	log(1, 0)	Error message
57			Negative answer	log(-1, 1)	Error message
58			Negative base	log(1, -1)	Error message
59	Abs	Valid	Positive integer	abs(4)	4
60			Positive decimal	abs(1.2)	1.2
61			Negative integer	abs(-4)	4
62			Negative decimal	abs(-1.2)	1.2

63	LCM	Valid	Positive integers	LCM(24, 32)	96
64		Valid Extreme	Positive integers	LCM(1, 1)	1
65			Zeros	LCM(0, 0)	Error message
66		Invalid	Decimals	LCM(1.2, 2.4)	Error message
67			Negatives	LCM(-1, -1)	Error message
68	HCF	Valid	Positive integers	HCF(24, 32)	8
69		Valid Extreme	Positive integers	HCF(1, 1)	1
70		Invalid Extreme	Zeros	HCF(0,0)	Error message
71		Invalid	Decimals	HCF(1.2, 2.4)	Error message
72			Negatives	HCF(-1,-1)	Error message
73	Quad	Valid	Integer coefficients, 2 distinct integer solutions	quadp(1, -3, 2)	2
74			Integer coefficients, 2 distinct integer solutions	quadn(1, -3, 2)	1
75			Integer coefficients, 1 repeated solution	quadp(1, 2, 1)	-1
76			Integer coefficients, 1 repeated solution	quadn(1, 2, 1)	-1
77			Decimal solution	quadp(1,-2,-2)	2.732…
78			Decimal solution	quadn(1,-2,-2)	-0.732…
79			Decimal coefficients	quadp(0.25,-0.5,-0.5)	2.732…
80			Decimal coefficients	quadp(0.25,-0.5,-0.5)	-0.732…
81		Invalid	Integer coefficients, no real solutions	quadp(1, 1, 1)	Error message
82			Integer coefficients, no real solutions	quadn(1, 1, 1)	Error message
83	Rand	Valid	Positive integers, correct order	rand(2,4)	2, 3 or 4
84		Valid	Negative integers, correct order	rand(-3,2)	-3, -2, -1, 0, 1 or 2
85		Invalid	Incorrect order	rand(2,1)	Error message
86		Invalid	Decimals	rand(0.2, 1)	Error message
87	Sine	Valid	Positive integer	sin(10)	-0.544…
88			Decimal	sin(1.2)	0.932…
89			Negative	sin(-1)	-0.841…
90	Cosine	Valid	Positive integer	cos(10)	-0.839…
91			Decimal	cos(1.2)	0.362…
92			Negative	cos(-1)	0.540…
93	Tangent	Valid	Positive integer	tan(10)	0.648…
94			Decimal	tan(1.2)	2.572…
95			Negative	tan(-1)	-1.557
96		Invalid	cos(x)=0	tan(pi/2)	Error message
97	Inverse sine	Valid	Within domain	arsin(0.5)	0.523…
98		Valid Extreme	1	arsin(1)	1.570…
99			-1	arsin(-1)	-1.570…
100			1.1	arsin(1.1)	Error message
101			-1.1	arsin(-1.1)	Error message
102		Invalid	invalid	arsin(10)	Error message

103	Inverse cosine	Valid	Within domain	arcos(0.5)	1.047…
104		Valid Extreme	1	arcos(1)	0
105			-1	arcos(-1)	3.141…
106		Invalid Extreme	1.1	arcos(1.1)	Error message
107			-1.1	arcos(-1.1)	Error message
108		Invalid	Invalid	arcos(10)	Error message
109	Inverse tangent	Valid	Positive integer	artan(10)	1.471…
110			Decimal	artan(1.2)	0.876…
111			Negative	artan(-1)	-0.785…
112	Hyperbolic sine	Valid	Positive integer	sinh(10)	11013.232…
113			Decimal	sinh(1.2)	1.509…
114			Negative	sinh(-1)	-1.175…
115	Hyperbolic cosine	Valid	Positive integer	cosh(10)	11013.232…
116			Decimal	cosh(1.2)	1.810…
117			Negative	cosh(-1)	1.543…
118	Hyperbolic tangent	Valid	Positive integer	tanh(10)	0.999…
119			Decimal	tanh(1.2)	0.833…
120			Negative	tanh(-1)	-0.761…
121	Inverse hyperbolic sine	Valid	Positive integer	arsinh(10)	2.998…
122			Decimal	arsinh(1.2)	1.015…
123			Negative	arsinh(-1)	-0.881…
124	Inverse hyperbolic cosine	Valid	Positive integer	arcosh(10)	2.993…
125			Decimal	arcosh(1.2)	0.622…
126		Valid Extreme	1	arcosh(1)	0
127		Invalid Extreme	0.9	arcosh(0.9)	Error message
128		Invalid	Negative	arcosh(-1)	Error message
129	Inverse hyperbolic tangent	Valid	0	artanh(0)	0
130		Valid Extreme	Upper	artanh(0.9)	1.472…
131			Lower	artanh(-0.9)	-1.472…
132		Invalid Extreme	Upper	artanh(1)	Error message
133			Lower	artanh(-1)	Error message
134		Invalid	Positive integer	artanh(10)	Error message
135	Constants	Valid	pi	pi	3.141…
136			tau	tau	6.283…
137			e	e	2.718…
138			g	g	9.806…
139			phi	phi	1.618…

Table 40: Test Plan for 7) Operations

Test 73: Quad Datatype Error

```
>quadp(1,-3,2)
Traceback (most recent call last):
  File "C:\Users\1892-FAMILY\Dropbox\Programming\Calculator\CodeBasics\7-Operations\Tested\Calc.py", line 324, in <module>
    print(calculate(expression))
  File "C:\Users\1892-FAMILY\Dropbox\Programming\Calculator\CodeBasics\7-Operations\Tested\Calc.py", line 307, in calculate
    expr = func(expr)
  File "C:\Users\1892-FAMILY\Dropbox\Programming\Calculator\CodeBasics\7-Operations\Tested\Calc.py", line 169, in tokenise
    tokens[-1] = tokens[-1].execute()
  File "C:\Users\1892-FAMILY\Dropbox\Programming\Calculator\CodeBasics\7-Operations\Tested\Datatypes.py", line 247, in execute
    return Num(self.__func(*self.__operands))
  File "C:\Users\1892-FAMILY\Dropbox\Programming\Calculator\CodeBasics\7-Operations\Tested\Operations.py", line 252, in func_quadp
    return (-b + (b**2 - 4*a*c)**0.5) / (2 * a)
TypeError: unsupported operand type(s) for ** or pow(): 'decimal.Decimal' and 'float'
```

The only ‘float’ on that line is the ‘0.5’ and the ‘decimal’ library struggles interacting with floats, so I need to convert it to the ‘Num’ datatype, so the ‘decimal’ library knows how to interact with it. This problem also applies to the negative square root version.

```python
def func_quadp(a, b, c):
    """Return the positive square root answer of the quadratic equation ax^2 + bx + c = 0"""

    from Datatypes import Num

    discriminant = b**2 - 4*a*c

    # invalid cases
    if discriminant < 0:
        raise CalcOperationError("No real solutions", "quadp", [a, b, c])

    # quadratic formula with positive square root
    return (-b + discriminant**Num("0.5")) / (2 * a)

def func_quadn(a, b, c):
    """Return the positive square root answer of the quadratic equation ax^2 + bx + c = 0"""

    from Datatypes import Num

    discriminant = b**2 - 4*a*c

    # invalid cases
    if discriminant < 0:
        raise CalcOperationError("No real solutions", "quadn", [a, b, c])

    # quadratic formula with negative square root
    return (-b - discriminant**Num("0.5")) / (2 * a)
```

The reason I have to import it only within the module is otherwise it leads to an import loop where one 'Datatypes.py' imports 'Operations.py' which would import 'Datatypes.py' and as global code executes when imported, I need to keep this local.

Retest:

```
>quadp(1,-3,2)
2.0000000000000000000000000000
```

Test 85: Rand Incorrect Order

```
>rand(2,1)
Traceback (most recent call last):
  File "C:\Users\james\Dropbox\Programming\Calculator\CodeBasics\7-Operations\Tested\Calc.py", line 324, in <module>
    print(calculate(expression))
  File "C:\Users\james\Dropbox\Programming\Calculator\CodeBasics\7-Operations\Tested\Calc.py", line 307, in calculate
    expr = func(expr)
  File "C:\Users\james\Dropbox\Programming\Calculator\CodeBasics\7-Operations\Tested\Calc.py", line 169, in tokenise
    tokens[-1] = tokens[-1].execute()
  File "C:\Users\james\Dropbox\Programming\Calculator\CodeBasics\7-Operations\Tested\Datatypes.py", line 247, in execute
    return Num(self.__func(*self.__operands))
  File "C:\Users\james\Dropbox\Programming\Calculator\CodeBasics\7-Operations\Tested\Operations.py", line 284, in func_rand
    return randint(low, high)
  File "D:\Program Files\Python364\lib\random.py", line 221, in randint
    return self.randrange(a, b+1)
  File "D:\Program Files\Python364\lib\random.py", line 199, in randrange
    raise ValueError("empty range for randrange() (%d,%d, %d)" % (istart, istop, width))
ValueError: empty range for randrange() (2,2, 0)
```

Empty range because the range *from* 2 *to* 1 includes nothing whereas the range *from* 1 *to* 2 includes 1 and 2. Rather than erroring, I will just switch them around:

```
def func_rand(low, high):
    """Return a random integer between 'low' and 'high'"""

    # invalid cases
    if low % 1 != 0 or high % 1 != 0:
        raise CalcOperationError("Must be whole numbers", "rand", [low, high])

    # if the wrong way around, swap them
    if low > high:
        low, high = high, low

    # use the function from the 'random' library
    return randint(low, high)
```

Retest:

```
>rand(2,1)
2
```

Testing Table

#	Actual Output	PASS/FAIL	Comment	Retest Output	Retest PASS/FAIL
1	2	PASS			
2	3.8	PASS			
3	Too few operands or too many operators	PASS			
4	1	PASS			
5	3.2	PASS			
6	2.98	PASS			

7	1	PASS			
8	1.7	PASS			
9	Too few operands or too many operators	PASS			
10	-1	PASS			
11	-2.3	PASS			
12	6	PASS			
13	0.45	PASS			
14	0	PASS			
15	2	PASS			
16	1.5	PASS			
17	While performing / with 2, 0: Cannot divide by 0	PASS			
18	2	PASS			
19	1	PASS			
20	While performing \ with 2, 0: Cannot divide by 0	PASS			
21	0	PASS			
22	2	PASS			
23	While performing % with 6, 0: Cannot divide by 0	PASS			
24	8	PASS			
25	0.5	PASS			
26	1.60…	PASS			
27	While performing ^ with 0, 0: 0 to the power of 0 is undefined	PASS			
28	1.41…	PASS			
29	1.22…	PASS			
30	While performing ¬ with -0.2, 2: The root must be a positive whole number	PASS			
31	While performing ¬ with 0, 2: The root must be a positive whole number	PASS			
32	6	PASS			
33	1	PASS			
34	1	PASS			
35	While performing P with 1.2, 2.2: Both must be whole numbers and cannot be negative	PASS			
36	-2	PASS	This is correct as permutations is executed before anything else, so it becomes '-(2P1)'		
37	Too few operands or too many operators	PASS			
38	While performing P with 1, 2: r (2) cannot be greater than n (1)	PASS			
39	3	PASS			
40	1	PASS			
41	1	PASS			

42	While performing C with 1.2, 2.2: Both must be whole numbers and cannot be negative	PASS			
43	-2	PASS	This is correct as combinations is executed before anything else, so it becomes '-(2C1)'		
44	Too few operands or too many operators	PASS			
45	6	PASS			
46	1	PASS			
47	While performing ! with 1.2: Must be whole number and cannot be negative	PASS			
48	-6	PASS	This is correct as the negative is executed after the factorial, so it becomes '-(3!)'		
49	0.693…	PASS			
50	0.182…	PASS			
51	While performing ln with 0: Can only find the natural log of positive numbers	PASS			
52	While performing ln with -1: Can only find the natural log of positive numbers	PASS			
53	3	PASS			
54	0.584…	PASS			
55	While performing log with 1, 0: Can only find the log of a positive number with a positive base	PASS			
56	While performing log with 0, 1: Can only find the log of a positive number with a positive base	PASS			
57	While performing log with 1, -1: Can only find the log of a positive number with a positive base	PASS			
58	While performing log with -1, 1: Can only find the log of a positive number with a positive base	PASS			
59	4	PASS			
60	1.2	PASS			
61	4	PASS			
62	1.2	PASS			
63	96	PASS			
64	1	PASS			
65	While performing LCM or HCF with 0: Must be a positive whole number	PASS			
66	While performing LCM or HCF with 1.2: Must be a positive whole number	PASS			

67	While performing LCM or HCF with -1: Must be a positive whole number	PASS			
68	8	PASS			
69	1	PASS			
70	While performing LCM or HCF with 0: Must be a positive whole number	PASS			
71	While performing LCM or HCF with 1.2: Must be a positive whole number	PASS			
72	While performing LCM or HCF with -1: Must be a positive whole number	PASS			
73	Crash	FAIL	The interaction between the decimal and float datatypes failed	2.00000…	PASS
74	1.00000…	PASS	When rounded, the trailing 0s will be removed		
75	-1	PASS			
76	-1	PASS			
77	2.732…	PASS			
78	-0.732…	PASS			
79	2.732…	PASS			
80	-0.732…	PASS			
81	While performing quadp with 1, 1, 1: No real solutions	PASS			
82	While performing quadn with 1, 1, 1: No real solutions	PASS			
83	4	PASS			
84	0	PASS			
85	Crash	FAIL	I forgot to check for if they're the wrong way around	2	PASS
86	While performing rand with 0.2, 1: Must be whole numbers	PASS			
87	-0.544…	PASS			
88	0.932…	PASS			
89	-0.841…	PASS			
90	-0.839…	PASS			
91	0.362…	PASS			
92	0.540…	PASS			
93	0.648…	PASS			
94	2.572…	PASS			
95	-1.557	PASS			
96	While performing tan with 1.570...: Tangent is undefined for values half way between multiples of pi	PASS			
97	0.523…	PASS			
98	1.570…	PASS			
99	-1.570…	PASS			

100	While performing arsin with 1.1: Inverse sine is only defined for values between -1 and 1 inclusive	PASS			
101	While performing arsin with -1.1: Inverse sine is only defined for values between -1 and 1 inclusive	PASS			
102	While performing arsin with 10: Inverse sine is only defined for values between -1 and 1 inclusive	PASS			
103	1.047…	PASS			
104	0	PASS			
105	3.141…	PASS			
106	While performing arcos with 1.1: Inverse cosine is only defined for values between -1 and 1 inclusive	PASS			
107	While performing arcos with -1.1: Inverse cosine is only defined for values between -1 and 1 inclusive	PASS			
108	While performing arcos with 10: Inverse cosine is only defined for values between -1 and 1 inclusive	PASS			
109	1.471…	PASS			
110	0.876…	PASS			
111	-0.785…	PASS			
112	11013.232…	PASS			
113	1.509…	PASS			
114	-1.175…	PASS			
115	11013.232…	PASS			
116	1.810…	PASS			
117	1.543…	PASS			
118	0.999…	PASS			
119	0.833…	PASS			
120	-0.761…	PASS			
121	2.998…	PASS			
122	1.015…	PASS			
123	-0.881…	PASS			
124	2.993…	PASS			
125	0.622…	PASS			
126	0	PASS			
127	While performing arcosh with 0.9: Inverse hyperbolic cosine is undefined for values less than 1	PASS			
128	While performing arcosh with -1: Inverse hyperbolic cosine is undefined for values less than 1	PASS			
129	0	PASS			
130	1.472…	PASS			
131	-1.472…	PASS			
132	While performing artanh with 1: Inverse hyperbolic tangent is only defined for values between -1 and 1 exclusive	PASS			

133	While performing artanh with -1: Inverse hyperbolic tangent is only defined for values between -1 and 1 exclusive	PASS			
134	While performing artanh with 10: Inverse hyperbolic tangent is only defined for values between -1 and 1 exclusive	PASS			
135	3.141…	PASS			
136	6.283…	PASS			
137	2.718…	PASS			
138	9.806…	PASS			
139	1.618…	PASS			

Table 41: Testing Table for 7) Operations

8) Settings

Using the command line interface in 'Calc.py':

#	Type	Description	Expression	Expected Output
1	Valid	Irrational number	pi	3.141592653589793
2	Valid	Normal	2^2	9
3	Valid	Exact sine values to test	sin(pi/6)	0.5
4	Valid		arsin(0.5)	0.523598775598299
5	Valid		pi/6	0.523598775598299

Table 42: Test Plan for 8) Settings

Some Successful Tests

```
>pi
3.141592653589793

>2^2
4

>sin(pi/6)
0.5

>arsin(0.5)
0.523598775598299

>pi/6
0.523598775598299
```

Testing Table

#	Actual Output	PASS/FAIL	Comment	Retest Output	Retest PASS/FAIL
1	3.141592653589793	PASS			
2	9	PASS			
3	0.5	PASS			
4	0.523598775598299	PASS			
5	0.523598775598299	PASS			

Table 43: Testing Table for 8) Settings

Evaluation

User Interviews

Questions

Out of 10, rate the following features in the calculator and explain why:

1) How intuitive error messages are
2) How clear the instructions are (wording not font)
3) How intuitive it is to navigate the GUI
4) How intuitive it is to enter expressions
5) How easy it is to read answers
6) How easy it is to read error messages
7) How easy it is to interact with memory

Interview 1 – my friend making the projectile modelling software

1) 8 – quite clear
2) 10 – very clear explanation but the text is messed up
3) 9 – labelled buttons make it easy
4) 7 – didn't realise I just had to type at first but now it's very easy
5) 9 – very clear
6) 9 – very clear
7) 7 – quite easy but doesn't always fit in the boxes

Interview 2 – a student from my maths class

1) 6 – some computing words I don't understand
2) 9 – clear explanation but some text overlaps
3) 10 – very easy
4) 8 – typing in the middle is good but I didn't know where I had to click at first
5) 10 – perfect
6) 7 – very easy but sometimes it overflows onto other text
7) 7 – could be more obvious that you can reference memory not shown in the list

Interview 3 – a student from my computing class

1) 8 – clear but some complex words
2) 10 – very clear
3) 10 – buttons make it clear
4) 9 – nice open space to enter the expressions
5) 10 – very clear
6) 8 – clear error messages but they sometimes go off the screen
7) 9 – very easy to use

Summary

Question	Interview			Average Score	
Number	1	2	3	Exact	Rounded
1	8	6	8	7.33	7
2	10	9	10	9.66	10
3	9	10	10	9.66	10
4	7	8	9	8.00	8
5	9	10	10	9.66	10
6	9	7	8	8.00	8
7	7	7	9	7.66	8

Table 44: User Interview GUI Scores

Objectives

1) I have included all operations I set out to include and all passed all of my thorough tests. This means all valid inputs lead to a correct answer and all invalid inputs lead to an error message, so I have met this objective.
2) I got an average score of 7 for intuitive error messages which does meet the objectives but only just. I asked my users why this isn't that high a score and they said the messages have some jargon in, such as 'invalid token', meaning they aren't as easy to understand as they could be.
3) Yes, 'Interface.py' does this whenever the calculator is called via it.
4) I have included both the constants I set out to plus 3 additional ones: tau, g and phi. The calculator can recall their exact value to over 30 decimal places although the rounding reduces this to 15. They can be used in operations correctly and all my tests passed.
5) I ran out of time to add settings and the ability to change them, so I have not met this objective.
6) Of all the sections I completed, my users gave them an average score of over 7/10 so I met this objective.
    a) Average rating: 10 for wording however the font wasn't very clear
    b) Average rating: 10
    c) Average rating: 8
    d) Average rating: 10
    e) Average rating: 8
    f) Average rating: 8
    g) I ran out of time to add settings so this could not be scored.

Apart from settings which I ran out of time for, my calculator has met all my objectives. The purpose for my calculator is to be a general scientific calculator and I think it does this even without settings.

Improvements

If I had more time or was to do this project again, there are several things I would add or change:

Settings

I would finish adding settings to my calculator in the 'post_calc' function in 'Calc.py'. I would also add a new screen in the GUI (like the instructions screen) to view and change settings. This would add more customisation to the calculator so users can use it how they like. This may make them more comfortable and be more likely to use it.

Exponentiation and unary operator precedence

As I discussed in test 30 of stage 1, this is a question that has no answer so brackets should be used if in any doubt. Handheld calculators put invisible brackets after all exponentiation operators meaning everything after them is in the power. However, they make this clear to the user by raising the power up as you would write it. My GUI cannot display things like we would write them so it wouldn't be clear enough what is in the power and what isn't. If I had more time, I would research this more and perhaps improve the GUI display so I could use invisible brackets.

Dynamically Adding Operations

If I had more time, I would add a JSON file to store the valid tokens and their functions which would be read into the calculator every time it is initialised. This would allow different programs to edit this JSON file and add new operations as well as write the code to execute these operations. I could then add a menu to the calculator to allow the user to add new operations which would be written to the JSON file.

GUI Text

Some text from the error message still overflows the screen and into the memory items sometimes. This only happens when the user types in a long string of characters with no spaces because I only start a newline at whitespace in the 'format_text' method of 'PygameTools.py'. The memory items in the list also are sometimes too long for the box so despite not overflowing, you can't see the whole thing so isn't always very useful.

The instructions screen font was very thick, making it difficult to read. I think this is because I am drawing the text on an intermediate surface before the final surface so perhaps it is compressed into an image before being drawn. If there is a way around it, this would require lots of research to fix.

GUI Symbol Representation

The cube root of 2 looks like $\sqrt[3]{2}$ in maths but in my calculator, it looks like '3¬2'. Similarly, in maths we normally write division as something 'over' something else with a horizontal line between them. This helps show which operations to do when, but in my calculator, everything is on 1 continuous line so it is harder to determine the order of operations which is less intuitive and can lead to the order of operations being different to what the user intended.

If I had more time, I would have written my own mathematical font that worked with Pygame so I could display these properly.

Appendix

Tables

Final Code

Below is the full, final, raw code for my project copied in with syntax highlighting. README.md is a markdown file which displays differently when opened properly.

Calc.py

```
"""
The calculator's core functionality
Use the 'calculate' function to calculate the answer to an expression
"""

from Datatypes import Stack, Queue, Operator, BothOperators, Num, OpenBracket, CloseBracket, valid_tokens,
regex, FunctionType, FunctionInstance
from Errors import CalcError
from decimal import DecimalException, Overflow, InvalidOperation

# instructions read from the text file for other files to import
with open("Instructions.txt", "r") as f:
    instructions = "".join(f.readlines())

def find_matched_key(match):
    """Return the key which was matched"""

    # exactly 1 value in 'match' will not be 'None'
    # so this will always return exactly once
    for key in match:
        if match[key] is not None:
            return key

def should_be_unary(prev_token):
    """Return whether or not the current token should be a unary operator depending on the previous token"""

    # if it is the first token in the expression, it should
    if prev_token is None:
        return True

    # if it's an open bracket, it should
    if isinstance(prev_token, OpenBracket):
        return True

    # if it's an operator, it should if it's binary or (unary and right associative)
    if isinstance(prev_token, Operator):

        # if it is binary, it should
        if not prev_token.is_unary:
            return True

        # if it is unary and right associative, it should
        if not prev_token.is_left_associative:
            return True

    # if none of the others are true, it shouldn't
    return False

def identify(name, value, prev_token):
    """Return an instance of a class to identify the token"""

    # if it's a number, convert my standard form notation into python's and make it an instance of 'Num'
    if name == "number":
        return Num(value.replace("~", "e"))

    # if it's a bracket, make it an instance of one of my bracket classes
    if value == "(":
        return OpenBracket()
    if value == ")":
        return CloseBracket()

    # if it's a comma, give an error message as commas can only be inside functions
    if value == ",":
        raise CalcError("Commas only allowed inside functions")

    # if it's in 'valid_tokens', it's a valid operator so:
    if value in valid_tokens:
        token = valid_tokens[value]

        # if it could be unary or binary, use the helper function to decide which and return that
        if isinstance(token, BothOperators):
```

```
            return token.unary if should_be_unary(prev_token) else token.binary

        # if it's a function, create a unique instance and return that
        if isinstance(token, FunctionType):
            return token.create(calculate)

        # otherwise just return it
        return token

    # otherwise, it is an invalid token so error
    raise CalcError("Invalid token: '{}'".format(value))

def identify_operand(operand, function):
    """Validate and format the operand of a function and store it in the function"""

    #remove whitespace at the start and end of the operand
    operand = operand.strip()

    # remove the '(' or ',' at the start of the operand or error if neither
    if operand[0] == "(" or operand[0] == ",":
        operand = operand[1:]
    else:
        raise CalcError("Functions must be immediately followed by brackets")

    # remove the ',' or ')' at the end of the operand or error if neither
    if operand[-1] == ")" or operand[-1] == ",":
        operand = operand[:-1]
    else:
        raise CalcError("Functions must end with a close bracket")

    # add the operand to the function object
    function.add_operand(operand)

def tokenise(expr):
    """Split the expression up into tokens and make them instances of classes to identify them"""

    # remove whitespace at the start or end of the expression and make lower case
    expr = expr.strip().lower()

    # initialise variables
    tokens = []
    pos = 0
    in_func = False

    while pos < len(expr):

        # find a regex match with the expression 'pos' characters in
        match = regex.match(expr, pos)

        # adjust 'pos' to be the end of the last match
        # so the next iteration doesn't match the same characters
        pos = match.end()

        # convert to a dictionary with the named patterns as keys
        match = match.groupdict()

        # find which key has been matched
        key = find_matched_key(match)

        # ignore whitespace
        if key != "whitespace":

            # if not in a function, identify the token and add it to the list of tokens
            if not in_func:

                # the previous token is the last token in the list 'tokens[-1]' but if the list is empty, it
is 'None'
                token = identify(key, match[key], tokens[-1] if tokens else None)
                tokens.append(token)

                # if the token is a function, initialise the function variables
                if isinstance(token, FunctionInstance):
                    in_func = True
                    bracket_depth = 0
                    operand_start_pos = pos

            # if in a function, ignore everything except brackets and commas
            # to get the operands as strings and add them to the function object
```

```
            else:
                if key == "bracket":

                    # if it's an open bracket, increase the bracket depth
                    if match[key] == "(":
                        bracket_depth += 1

                    # if it's a close bracket, decrease the bracket depth
                    else:
                        bracket_depth -= 1

                        # if the bracket depth is now 0, add the last operand and execute the function
                        if bracket_depth == 0:
                            in_func = False
                            identify_operand(expr[operand_start_pos:pos], tokens[-1])
                            tokens[-1] = tokens[-1].execute()

                # if it's a comma, add the operand to the function
                # and update the start pos for the next operand
                elif key == "comma" and bracket_depth == 1:
                    identify_operand(expr[operand_start_pos:pos], tokens[-1])
                    operand_start_pos = pos - 1

                # add omitted closing brackets around functions
                elif pos >= len(expr):
                    expr += ")"

    return tokens

def should_be_executed_first(top_of_stack_token, current_token):
    """Return whether or not the token at the top of the stack should be executed before the current token"""

    # the token at the top of the stack should be executed before the current token if 1 and 2:
    #   1) the token at the top of the stack is an operator
    #   2) if a or b:
    #       a) the operator at the top of the stack has better precedence than the current operator
    #       b) i and ii:
    #           i)  they have equal precedences
    #           ii) the operator at the top of the stack is left associative

    #      ------------------ 1 ------------------- and -----------------------------------------------------
--------------------------------- 2 ---------------------------------------------------------------------
--
    #                                                -------------------------- a ----------------------
---- or ---------------------------------------------------- b ----------------------------------------------
-
    #
----------------------------- i -------------------------- and ------------------ ii ----------------
    return isinstance(top_of_stack_token, Operator) and (top_of_stack_token.precedence <
current_token.precedence or (top_of_stack_token.precedence == current_token.precedence and
top_of_stack_token.is_left_associative))

def convert(tokens):
    """Convert the tokens from infix notation to postfix notation (AKA reverse polish notation) by the
shunting yard algorithm"""

    output_queue = Queue()      # we only ever add numbers or operators to the output queue
    operator_stack = Stack()    # we only ever add operators and open brackets to the operator stack

    for token in tokens:

        # add numbers to the output queue
        if isinstance(token, Num):
            output_queue.enqueue(token)

        # if it's an operator, add any operators on the stack that should be executed before
        # it (using 'should_be_executed_first') to the output queue and then add it to the stack
        elif isinstance(token, Operator):
            while should_be_executed_first(operator_stack.peek(), token):
                output_queue.enqueue(operator_stack.pop())
            operator_stack.push(token)

        # add open brackets to the operator stack
        elif isinstance(token, OpenBracket):
            operator_stack.push(token)

        # otherwise, it must be a close bracket so add everything from the operator stack to the output queue
        # until there is an open bracket, then remove this too but don't add it to the output queue
```

```
        else:
            while not isinstance(operator_stack.peek(), OpenBracket) and operator_stack.peek() is not None:
                output_queue.enqueue(operator_stack.pop())

            # if there is nothing left in the stack but we haven't
            # found an open bracket, there are too few open brackets
            if operator_stack.peek() is None:
                raise CalcError("Too many close brackets or not enough open brackets")

            # remove the open bracket from the stack and discard
            else:
                operator_stack.pop()

    # move everything on the stack to the output queue
    while operator_stack:
        token = operator_stack.pop()

        # if an open bracket didn't have a matching closing bracket,
        # it could appear here but we don't want it so ignore it
        if not isinstance(token, OpenBracket):
            output_queue.enqueue(token)

    return output_queue

def execute(queue):
    """
    Execute the tokens to get a final answer
    As in postfix notation, the first operator in the queue is the first operator to be executed
    so execute this with the operands repeatedly until all of them have been executed to get a final answer
    """

    stack = Stack()

    for token in queue:

        # if it's a number, push it to the stack
        if isinstance(token, Num):
            stack.push(token)

        # otherwise it must be an operator so pop its operands from the stack, execute it with them and add
push the result to the stack
        else:

            # at least 1 operand is needed for all operators
            operands = [stack.pop()]

            # binary operators need another
            if not token.is_unary:
                operands.append(stack.pop())

            # the stack returns None if empty so if 'None' is in there, there are too few operands
            if None in operands:
                raise CalcError("Too few operands or too many operators")

            # execute the operator with its operands (reversed)
            # catch errors raised by the 'decimal' library and convert them to my format
            try:
                token = token.execute(operands[::-1])
            except InvalidOperation:
                raise CalcError("Invalid operation")
            except Overflow:
                raise CalcError("Number too big")
            except DecimalException as e:
                raise CalcError("Error: " + str(e).split("decimal.")[1].split("'>]")[0])

            # make it a 'Num' and push it to the stack
            stack.push(Num(token))

    # there should be exactly 1 number on the stack at the end - the answer
    # if not, there are too many operands or too few operators
    if len(stack) != 1:
        raise CalcError("Too many operands or too few operators")

    # the last item on the stack is the answer
    return stack.pop()

def post_calc(ans):
    """Apply settings to the answer"""
```

```
    # check a valid number
    ans = float(ans)
    if ans == float("inf"):
        raise CalcError("Number too big")

    # round all answers to 15 decimal places
    ans = round(ans, 15)

    # convert to a string so it is in an exact and built-in type
    ans = str(ans)

    # remove trailing 0s
    while ans[-1] == "0":
        ans = ans[:-1]

    # remove the decimal point if it ends in one but nothing after that
    if ans[-1] == ".":
        ans = ans[:-1]

    # if the number is now '-0', make '0'
    if ans == "-0":
        ans = "0"

    # convert back to my representation of standard form
    ans = ans.lower().replace("e", "~")

    return ans

def calculate(expr, debug=False):
    """
    Calculate the answer to 'expr'.
    If CalcError (or it's child CalcOperationError) has been raised,
    it is due to an invalid expression so needs to be caught and presented as an error message
    Any other exceptions are errors in the code

    :param expr (str): The expression to execute
    :param debug (bool): Whether or not to print out extra information to check for errors. Default: False
    :return ans (str): The answer to 'expr'
    """

    assert isinstance(expr, str), "param 'expr' must be a string"

    for func in [tokenise, convert, execute, post_calc]:

        # execute each function with the result from the last
        expr = func(expr)

        # output the progress if debug is on
        if debug:
            print(str(func).split("function ")[1].split(" at ")[0] + ":", repr(expr))

    return expr

# only runs if the file is run directly (not if imported)
if __name__ == "__main__":

    # quick interface to test the calculator with debug on to check it's working
    # catches errors due to the user's input and displays them as error messages
    # repeats until the user enters an empty expression
    expression = input("\n>")
    while expression != "":
        try:
            print(calculate(expression))
        except CalcError as e:
            print(e)
        expression = input("\n>")
```

Datatypes.py

```
"""
Contains datatypes the calculator needs to function as part of its internal operation
as well as the list of all valid tokens and the regex pattern to tokenise expressions
"""

from re import VERBOSE, compile as compile_regex
from collections import deque
from Operations import op_pos, op_add, op_neg, op_sub, op_mul, op_true_div, op_floor_div, op_mod, op_exp,
op_root, op_permutations, op_combinations, op_factorial, func_ln, func_log, func_abs, func_lcm, func_hcf,
```

```
func_rand, func_quadp, func_quadn, func_sin, func_cos, func_tan, func_arsin, func_arcos, func_artan,
func_sinh, func_cosh, func_tanh, func_arsinh, func_arcosh, func_artanh
from decimal import Decimal
from Errors import CalcError

class Stack:
    """Represents a stack - a LIFO data structure similar to a list/array with only access to the top"""

    def __init__(self):
        # private attribute denoted by the double underscore prefix
        self.__stack = deque()

    def push(self, item):
        """Add 'item' to the top of the stack"""
        self.__stack.append(item)

    def pop(self):
        """Remove and return the item on the top of the stack. Return 'None' if the stack is empty"""
        return self.__stack.pop() if self else None

    def peek(self):
        """Return the item on the top of the stack without removing it from the stack. Return 'None' if the
stack is empty"""
        return self.__stack[-1] if self else None

    def __bool__(self):
        return bool(self.__stack)

    def __len__(self):
        return len(self.__stack)

    def __repr__(self):
        return "Stack({})".format(str(list(self.__stack)).replace("[", "").replace("]", ""))

    def __iter__(self):

        # need to copy it because the variable is a pointer to where the actual stack is so
        # creating a new variable will mean they both reference the same thing whereas now
        # they reference identical but different stacks which is what I want because I am removing
        # items from the copy which I don't want to happen to the real stack
        temp = self.__stack.copy()

        while temp:
            yield temp.pop()

    def __contains__(self, item):

        for i in self:
            if i == item:
                return True

        return False

class Queue:
    """Represents a queue - a FIFO data structure similar to a list/array with only access to the head and
tail"""

    def __init__(self):
        # private attribute denoted by the double underscore prefix
        self.__queue = deque()

    def enqueue(self, item):
        """Add 'item' to the tail of the queue"""
        self.__queue.appendleft(item)

    def dequeue(self):
        """Remove and return the item at the head of the queue. Return 'None' if the queue is empty"""
        return self.__queue.pop() if self else None

    def peek(self):
        """Return the item at the head of the queue without removing it from the queue. Return 'None' if the
queue is empty"""
        return self.__queue[-1] if self else None

    def __bool__(self):
        return bool(self.__queue)

    def __len__(self):
```

```
        return len(self.__queue)

    def __repr__(self):
        return "Queue({})".format(str([item for item in self]).replace("[", "").replace("]", ""))

    def __iter__(self):

        # need to copy it because the variable is a pointer to where the actual queue is so
        # creating a new variable will mean they both reference the same thing whereas now
        # they reference identical but different queues which is what I want because I am removing
        # items from the copy which I don't want to happen to the real queue
        temp = self.__queue.copy()

        while temp:
            yield temp.pop()

    def __contains__(self, item):

        for i in self:
            if i == item:
                return True

        return False

class Operator:
    """
    Represents an operator and stores information about it

    :param name (str): The name of the operator
    :param func (identifier): The identifier of the function to execute the operation
    :param precedence (int/float): The precedence of the operation compared to other operations. Lower
numbers means executed first
    :param is_left_associative (bool): Whether or not the operator is left (-to-right) associative (right (-
to-left) associative otherwise)
    :param is_unary (bool): Whether or not the operator is a unary operator (takes only 1 operand) or
otherwise it is binary (takes 2 operands)
    """

    def __init__(self, name, func, precedence, is_left_associative, is_unary):
        self.name = name
        self.func = func
        self.precedence = precedence
        self.is_left_associative = is_left_associative
        self.is_unary = is_unary

    def execute(self, operands):
        """
        Return the answer when the operator is executed with its operands

        :param operands (list): The operands to execute the operator with
        :return answer (int/float): The answer when the operator is executed with the operands
        """

        # the star splits the 'operands' list out into individual parameters
        return self.func(*operands)

    def __repr__(self):
        return "UnaryOperator({})".format(self.name) if self.is_unary else
"BinaryOperator({})".format(self.name)

class BinaryOperator(Operator):
    """
    Represents a binary operator and stores information about it

    :param name (str): The name of the operator
    :param func (identifier): The identifier of the function to execute the operation
    :param precedence (int/float): The precedence of the operation compared to other operations. Lower
numbers means executed first
    :param is_left_associative (bool): Whether or not the operator is left (-to-right) associative
(alternative is right (-to-left) associative)
    """

    def __init__(self, name, func, precedence, is_left_associative):
        super().__init__(name, func, precedence, is_left_associative, False)

class UnaryOperator(Operator):
    """
    Represents a unary operator and stores information about it
```

```
    :param name (str): The name of the operator
    :param func (identifier): The identifier of the function to execute the operation
    :param is_left_associative (bool): Whether or not the operand is on the left side of the operator
(alternative is on the right)
    """

    def __init__(self, name, func, is_left_associative):
        super().__init__(name, func, 1, is_left_associative, True)

class BothOperators:
    """
    Represents a symbol that could represent a binary operator or a unary operator.

    :param unary (UnaryOperator): The unary operator that it could be
    :param binary (BinaryOperator): The binary operator that it could be
    """

    def __init__(self, unary, binary):
        assert isinstance(unary, Operator) and unary.is_unary, "Must be an instance of 'Operator' and be
unary"
        assert isinstance(binary, Operator) and not binary.is_unary, "must be an instance of 'Operator' and
be binary"
        self.unary = unary
        self.binary = binary

    def __repr__(self):
        return "BothOperators({}, {})".format(self.unary, self.binary)

class FunctionType:
    """
    Represents a type of function and stores information about it

    :param name (str): The name of the type of function
    :param func (identifier): The identifier of the function to execute the operation
    :param num_operands (int): The number of operands the function takes
    """

    def __init__(self, name, func, num_operands):
        self.__name = name
        self.__func = func
        self.__num_operands = num_operands

    def create(self, calc):
        """
        Return a new object which has the same properties as this object
        but is unique for all instances of the function in the expression

        :param calc (function): The calculate function from the main calculator
        :return (object): An instance of the 'FunctionInstance' class
        """

        return FunctionInstance(self.__name, self.__func, self.__num_operands, calc)

    def __repr__(self):
        return "FunctionType({})".format(self.__name)

class FunctionInstance:
    """
    Represents a function instance and stores information about it

    :param name (str): The name of the type of function
    :param func (function): The function to execute the operation
    :param num_operands (int): The number of operands the function takes
    :param calc (function): The calculate function from the main calculator
    """

    def __init__(self, name, func, num_operands, calc):
        self.__name = name
        self.__func = func
        self.__num_operands = num_operands
        self.__calc = calc
        self.__operands = []

    def add_operand(self, operand):
        """
        Execute the operand with the calculator to simplify it and then add it to the stored operands
```

```
        :param operand (num): The operand to add
        """

        self.__operands.append(Num(self.__calc(operand)))

    def execute(self):
        """
        Recursively execute all operands using the calculator and then
        return the answer when the function is executed with its operands

        :return (Num): The answer to the function when executed on its operands
        """

        if len(self.__operands) != self.__num_operands:
            raise CalcError("{} operands required in {} function call".format(self.__num_operands,
self.__name))

        # execute the function with its operands and return it
        # the star splits the list out into individual arguments
        return Num(self.__func(*self.__operands))

    def __repr__(self):
        return "{}({})".format(self.__name, ", ".join([str(operand) for operand in self.__operands]))

class Num(Decimal):
    """Represents a number"""
    # inherits all methods from Decimal but overrides representation method
    def __repr__(self):
        return "Num({})".format(self)

class Bracket:
    """
    Represents a bracket

    :param is_open (bool): Whether or not the bracket is an open bracket (alternative is a close bracket)
    """

    def __init__(self, is_open):
        self.is_open = is_open

    def __repr__(self):
        return "OpenBracket" if self.is_open else "CloseBracket"

class OpenBracket(Bracket):
    """Represents an open bracket"""

    def __init__(self):
        super().__init__(True)

class CloseBracket(Bracket):
    """Represents a close bracket"""

    def __init__(self):
        super().__init__(False)

# create the tokens that can be used in the calculator with the classes above and the operations in
'Operations.py'
# the key is the symbol that will be in expressions and the value is an instance of one of the following
classes:
# UnaryOperator, BinaryOperator, Operator, or BothOperators
valid_tokens = {
    "+": BothOperators(UnaryOperator("Positive (+)", op_pos, False), BinaryOperator("Addition (+)", op_add,
4, True)),
    "-": BothOperators(UnaryOperator("Negative (-)", op_neg, False), BinaryOperator("Subtraction (-)",
op_sub, 4, True)),
    "*": BinaryOperator("Multiplication (*)", op_mul, 3, True),
    "/": BinaryOperator("Division (/)", op_true_div, 3, True),
    "\\": BinaryOperator("Floor division (\\)", op_floor_div, 3, True),
    "%": BinaryOperator("Mod (%)", op_mod, 3, True),
    "^": BinaryOperator("Exponentiation (^)", op_exp, 2, False),
    "¬": BinaryOperator("Root (¬)", op_root, 2, False),
    "p": BinaryOperator("Permutations (P)", op_permutations, 0, True),
    "c": BinaryOperator("Combinations (C)", op_combinations, 0, True),
    "!": UnaryOperator("Factorial (!)", op_factorial, True),
    "ln": FunctionType("Natural log (ln)", func_ln, 1),
    "log": FunctionType("Logarithm (log)", func_log, 2),
    "abs": FunctionType("Absolute value (abs)", func_abs, 1),
    "lcm": FunctionType("Lowest common multiple", func_lcm, 2),
```

```
    "hcf": FunctionType("Highest common factor", func_hcf, 2),
    "rand": FunctionType("Random number generator", func_rand, 2),
    "quadp": FunctionType("Quadratic equation solver (postive square root)", func_quadp, 3),
    "quadn": FunctionType("Quadratic equation solver (negative square root)", func_quadn, 3),
    "sin": FunctionType("Sin (sin)", func_sin, 1),
    "cos": FunctionType("Cosine (cos)", func_cos, 1),
    "tan": FunctionType("Tangent (tan)", func_tan, 1),
    "arsin": FunctionType("Inverse sine (arsin)", func_arsin, 1),
    "arcos": FunctionType("Inverse cosine (arcos)", func_arcos, 1),
    "artan": FunctionType("Inverse tangent (artan)", func_artan, 1),
    "sinh": FunctionType("Hyperbolic sin (sinh)", func_sinh, 1),
    "cosh": FunctionType("Hyperbolic cosine (cosh)", func_cosh, 1),
    "tanh": FunctionType("Hyperbolic tangent (tanh)", func_tanh, 1),
    "arsinh": FunctionType("Inverse hyperbolic sine (arsinh)", func_arsinh, 1),
    "arcosh": FunctionType("Inverse hyperbolic cosine (arcosh)", func_arcosh, 1),
    "artanh": FunctionType("Inverse hyperbolic tangent (artanh)", func_artanh, 1),
    "pi": Num("3.14159265358979323846264338327950288"),
    "tau": Num("6.28318530717958647692528676655900576"),
    "e": Num("2.71828182845904523536028747135266249"),
    "g": Num("9.80665"),
    "phi": Num("1.61803398874989484820458683436563811")
}

# compile the regex pattern that will be used to check for tokens
regex = compile_regex(r"""
    (?P<whitespace>\s+)
    |(?P<number>(\d*\.)?\d+(~[+-]?\d+)?)
    |(?P<word>[a-z]+)
    |(?P<bracket>[()])
    |(?P<comma>,)
    |(?P<other>.)
""", VERBOSE)
```

Errors.py

```
"""
Contains custom error types that could occur in the calculator
When the error is in the expression, raise the one of these exceptions
This means it can be distinguished from errors in code, caught and presented as an error message without
crashing
Make error messages as clear as possible so the user can understand them
"""

class CalcError(Exception):
    """General errors in the calculator"""
    # inherits all of 'Exception's methods and attributes
    pass

class CalcOperationError(CalcError):
    """
    Errors in the calculator due to specific requirements of operations
    eg: factorial is undefined for negatives or decimals

    :param msg (str): The error message - as clear as possible so someone with no mathematical or programming
knowledge can understand
    :param op_name (str): The name of the operation that errored
    :param operands (iterable): The operands the operation was called with
    """
    # inherits all of 'CalcError' and therefore 'Exception's method and attributes but overrides the init
method
    def __init__(self, msg, op_name, operands):
        # converts all operands to strings and separates them by commas in the message
        super().__init__("While performing {} with {}: {}".format(op_name, ", ".join([str(operand) for
operand in operands]), msg))
```

Instructions.txt

Enter an expression to calculate the answer.

Operators are represented by a symbol and perform an operation on the numbers around them. Binary operators have 2 numbers (1 either side of the symbol), whereas unary operators have 1 number (either left or right of the symbol). Functions are represented by a word followed by brackets containing all operands (values needed for the function) separated by commas. Constants are a word representing a number very accurately. Simply enter the word and it will convert it to the number.

Functions and constants are always executed first and then operators are executed using BODMAS - brackets, other (exponents and unary operators), division and multiplication, addition and subtraction.

Binary Operators:

Addition: use '+' between 2 numbers to find the first add the second.
Subtraction: use '-' between 2 numbers to find the first subtract the second.
Multiplication: use '*' between 2 numbers to find the first multiplied by the second.
Division (true): use '/' between 2 numbers to find the first divided by the second.
Division (floor): use '\' between 2 numbers to find the first divided by the second and rounded down to the nearest whole number.
Mod: use '%' between 2 numbers to find the remainder after the first is divided by the second.
Exponentiation: use '^' between 2 numbers to find the first to the power of the second.
Root: use '¬' to find the first th root of the second - '2¬a' is the square root of 'a', '3¬a' is the cube root of 'a', etc.
Permutations: use 'P' to find the number of ways there are to organise the second number of items into the first number of places including all possible orders.
Combinations: use 'C' to find the number of ways there are to organise the second number of items into the first number of places only counting 1 possible order.

Unary Operators:
Positive: use '+' before a number to find the positive of it.
Negative: use '-' before a number to find the negative of it.
Factorial: use '!' after a positive whole number to find the product of all positive whole numbers less than or equal to it.

Functions:
Natural log: use 'ln' with 1 operand to find the natural logarithm of it.
Logarithm: use 'log' with 2 operands to find the logarithm of the first to the second base.
Absolute value: use 'abs' with 1 operand to find the absolute value of it which is always positive.
Lowest common multiple: use 'lcm' with 2 operands to find the lowest common multiple of them.
Highest common factor: use 'hcf' with 2 operands to find the highest common factor of them.
Random number generator: use 'rand' with 2 operands to find a random integer between them, inclusive.
Quadratic equation solver: use 'quadp' with 3 operands (a, b and c) to find the positive square root answer to the quadratic equation 'ax^2 + bx + c = 0' or use 'quadn' to find the negative square root answer of the same equation.
Sine: use 'sin' with 1 operand (an angle) to find the ratio between the opposite side and hypotenuse of its triangle.
Cosine: use 'cos' with 1 operand (an angle) to find the ratio between the adjacent side and hypotenuse of its triangle.
Tangent: use 'tan' with 1 operand (an angle) to find the radio between the opposite and adjacent sides of its triangle.
Inverse sine: use 'arsin' with 1 operand between -1 and 1 inclusive to find the angle it makes with the opposite side and hypotenuse of its triangle.
Inverse cosine: use 'arcos' with 1 operand between -1 and 1 inclusive to find the angle it makes with the adjacent side and hypotenuse of its triangle.
Inverse tangent: use 'artan' with 1 operand to find the angle it makes with the opposite and adjacent sides of its triangle.
Hyperbolic sine: use 'sinh' with 1 operand (an angle) to find the ratio between the opposite side and hypotenuse of its hyperbola.
Hyperbolic cosine: use 'cosh' with 1 operand (an angle) to find the ratio between the adjacent side and hypotenuse of its hyperbola.
Hyperbolic tangent: use 'tanh' with 1 operand (an angle) to find the ratio between the opposite and adjacent sides of its hyperbola.
Inverse hyperbolic sine: use 'arsinh' with 1 operand to find the angle it makes with the opposite side and hypotenuse of its hyperbola.
Inverse hyperbolic cosine: use 'arcosh' with 1 operand at least 1 to find the angle it makes with the adjacent side and hypotenuse of its hyperbola.
Inverse hyperbolic tangent: use 'artanh' with 1 operand between -1 and 1 inclusive to find the angle it makes with the opposite and adjacent sides of its hyperbola.

Constants:
pi: use 'pi' to get the ratio between a circle's circumference and its diameter. Value = 3.1415...
tau: use 'tau' to get 2 lots of pi - the number of radians in 360 degrees. Value = 6.2831...
e: use 'e' to get Euler's number. Value = 2.7182...
g: use 'g' to get the acceleration due to gravity close to the Earth's surface. Value = 9.80665
phi: use 'phi' to get the golden ratio found in many places in nature. Value = 1.6180...

Interface.py

```
"""
Contains the interface between a user interface and the calculator

User interfaces should use the calculator via this to record and provide access to memory by instantiating
the 'Interface' class and:
- if the user wants to calculate the answer to an expression, use the 'calculate' method
- if the user wants to view instructions, use the 'instructions' attribute
- if the user wants to view memory, use the 'recent_memory' method
- if the user wants to clear memory, use the 'clear_memory' method
- if the user wants to insert a specific memory answer into their expression:
    1) get which memory item is being requested
    2) use the 'memory_item' method to get the original expression and answer of interest
    3) it may be best to re-calculate the answer using the original expression and the 'calculate' method
    4) insert the answer from memory or the re-calculated answer into the expression

Before calling the 'recent_memory' or 'memory_item' methods with a number from the user,
```

```
the interface should call 'len_memory' to check how many items are in memory and verify the number wanted
is a valid number and equal to or less than the number of items in memory. If not, display the relevant error
message
If either of these methods are called with invalid parameters, they will raise 'IndexError'
"""

from Calc import calculate, instructions
from Errors import CalcError

class Interface:
    """
    The interface between a user interface and the calculator
    Stores and allows access to memory
    """

    def __init__(self):

        # private attributes
        self.__memory = []
        self.__instructions = instructions

    @property
    def instructions(self):
        """Return the instructions without being able to change it"""
        return self.__instructions

    def calculate(self, expr):
        """
        Calculate the answer to 'expr', storing the expression and answer in memory for later recall
        If CalcError (or it's child CalcOperationError) has been raised,
        it is due to an invalid expression so needs to be caught and presented as an error message
        Any other exceptions are errors in the code

        :param expr (str): The expression to execute
        :return ans (str): The answer to 'expr'
        """

        # calculate the answer with the calculator
        ans = calculate(expr)

        # add the expression and answer to the front of the list
        self.__memory.insert(0, (expr, ans))

        return ans

    def len_memory(self):
        """
        Return the number of items in memory

        :return (int): The number of items in memory
        """

        return len(self.__memory)

    def memory_item(self, num_calculations_ago=1):
        """
        Retrieve an answer from memory along with the expression that resulted in it
        Will raise 'IndexError' if 'num_calculations_ago' isn't an integer between 1 and the number of items
in memory

        :param num_calculations_ago (int): The item to retrieve from memory: 1 is the most recent
calculation, ascending from there. Default: None
        :return (tuple): A 2-value tuple where the 0th index is the string expression and the 1st is the
string answer
        """

        # invalid cases
        if not isinstance(num_calculations_ago, int):
            raise IndexError("Must be an integer")
        if self.len_memory() < num_calculations_ago:
            raise IndexError("There aren't that many items saved in memory")
        if num_calculations_ago < 1:
            raise IndexError("Must be greater than or equal to 1")

        # typical cases
        return self.__memory[num_calculations_ago - 1]

    def recent_memory(self, num_to_retrieve=None):
```

```
        """
        Retrieve a list of answers from memory along with the expressions that resulted in each of them
        Will raise IndexError if 'num_to_retrieve' isn't an integer greater than or equal to 1

        :param num_to_retrieve (int): The number of answers to retrieve. 'None' means all. Default: None
        :return (list): The memory items (most recent first) which are each a 2-value tuple where the 0th
                        index is the string expression and the 1st is the string answer
        """

        # make 'None' mean all and if there are less than asked for, just return the number available
        if num_to_retrieve is None or num_to_retrieve > self.len_memory():
            num_to_retrieve = self.len_memory()

        # invalid cases
        if not isinstance(num_to_retrieve, int):
            raise IndexError("Must be an integer (whole number")
        if num_to_retrieve < 0:
            raise IndexError("Must be greater than or equal to 0")

        # typical cases
        # the slice selects the first 'num_to_retrieve' items in the list 'self.__memory'
        return self.__memory[:num_to_retrieve]

    def clear_memory(self):
        """Clear the calculator's memory"""

        self.__memory.clear()

# only runs if the file is run directly (not if imported)
if __name__ == "__main__":

    # instantiate the class
    calc = Interface()

    # extend instructions
    new_instructions = calc.instructions + "\n\nType 'memory' to view previous calculations, 'instructions'
to view instructions, 'clear' to clear the memory, 'ans' in place of the previous answer and 'Mx' where x is
a number in place of the xth previous answer"

    # quick user interface to test the calculator through the interface and memory access
    # repeats until the user enters an empty expression
    expression = input("\n>").lower()
    while expression != "":

        # typing 'instructions' will output the general instructions
        if "instructions" in expression:
            print(new_instructions)

        # typing 'clear' will clear the memory
        elif "clear" in expression:
            calc.clear_memory()
            print("Memory cleared")

        # typing 'memory' will output all previous calculations
        elif "memory" in expression:
            count = 1
            for expr, ans in calc.recent_memory():
                print("{}: {} = {}".format(count, expr, ans))
                count += 1
            if count == 1:
                print("Memory is empty")

        else:
            try:
                # including 'ans' will replace it with the previous answer
                if "ans" in expression:
                    if calc.len_memory() == 0:
                        raise CalcError("Memory is empty")
                    else:
                        expression = expression.replace("ans", calc.memory_item()[1])

                # including 'Mx' will replace it with the xth previous answer
                while "m" in expression:

                    # take all digits between the 'M' and the first non-digit
                    index = expression.split("m")[1]
                    pos = 0
                    while pos < len(index) and index[pos] in "0123456789":
```

```
                    pos += 1
                index = index[:pos]

                if index == "":
                    raise CalcError("You must give a memory reference after 'm'")

                try:
                    index = int(index)
                except ValueError:
                    raise CalcError("Memory references must be whole numbers")
                else:
                    if index > calc.len_memory():
                        raise CalcError("Not enough items in memory")
                    elif index < 1:
                        raise CalcError("Memory references must be greater than or equal to 1")
                expression = expression.replace("m{}".format(index), calc.memory_item(index)[1])

            # calculate the answer
            print(calc.calculate(expression))

        # catch and output errors
        except CalcError as e:
            print(e)

    expression = input("\n>").lower()
```

Operations.py

```
"""
Contains the code for the operations that can be used in the calculator
"""

from Errors import CalcOperationError
from math import log, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh
from random import randint

def op_add(x, y):
    """Return x add y"""

    # typical cases
    return x + y

def op_sub(x, y):
    """Return x subtract y"""

    # typical cases
    return x - y

def op_mul(x, y):
    """Return x multiplied by y"""

    # typical cases
    return x * y

def op_true_div(x, y):
    """Return x divided by y"""

    # invalid case
    if y == 0:
        raise CalcOperationError("Cannot divide by 0", "/", [x, y])

    # typical cases
    return x / y

def op_floor_div(x, y):
    """Return x divided by y, rounded down"""

    # invalid case
    if y == 0:
        raise CalcOperationError("Cannot divide by 0", "\\", [x, y])

    # typical cases
    return x // y

def op_mod(x, y):
    """Return x mod y - the remainder when x is divided by y"""

    # invalid case
    if y == 0:
```

```
        raise CalcOperationError("Cannot divide by 0", "%", [x, y])

    # typical cases
    return x % y

def op_exp(x, y):
    """Return x to the power of y"""

    # invalid case
    if x == 0 and y == 0:
        raise CalcOperationError("0 to the power of 0 is undefined", "^", [x, y])

    # typical cases
    return x ** y

def op_root(root, x):
    """Return the root th root of x"""

    # invalid cases
    if root <= 0 or root % 1 != 0:
        raise CalcOperationError("The root must be a positive whole number", "¬", [root, x])

    # specific case of power
    return x ** (1 / root)

def op_permutations(n, r):
    """Return the number of ways there are to arrange r things in n places, counting all orders"""

    # invalid cases
    if n % 1 != 0 or r % 1 != 0 or n < 0 or r < 0:
        raise CalcOperationError("Both must be whole numbers and cannot be negative", "P", [n, r])
    if r > n:
        raise CalcOperationError("r ({}) cannot be greater than n ({})".format(r, n), "P", [n, r])

    # use the factorial operation already defined
    return op_factorial(n) / op_factorial(n - r)

def op_combinations(n, r):
    """Return the number of ways there are to arrange r things in n places, only counting 1 order"""

    # invalid cases
    if n % 1 != 0 or r % 1 != 0 or n < 0 or r < 0:
        raise CalcOperationError("Both must be whole numbers and cannot be negative", "C", [n, r])
    if r > n:
        raise CalcOperationError("r ({}) cannot be greater than n ({})".format(r, n), "C", [n, r])

    # use the factorial operation already defined
    return op_factorial(n) / (op_factorial(r) * op_factorial(n - r))

def op_pos(x):
    """Return the positive of x"""

    # typical cases
    return +x

def op_neg(x):
    """Return the negative of x"""

    # typical cases
    return -x

def op_factorial(x):
    """Return x factorial"""

    # invalid cases
    if x < 0 or x % 1 != 0:
        raise CalcOperationError("Must be whole number and cannot be negative", "!", [x])

    # multiply all numbers between 1 and x together
    product = 1
    while x > 1:
        product *= x
        x -= 1

    return product

def func_ln(x):
    """Return ln(x)"""
```

```
    # invalid cases
    if x <= 0:
        raise CalcOperationError("Can only find the natural log of positive numbers", "ln", [x])

    # use the function from the 'math' library
    return log(x)

def func_log(x, base):
    """Return log of x to the base 'base'"""

    # invalid cases
    if x <= 0 or base <= 0:
        raise CalcOperationError("Can only find the log of a positive number with a positive base", "log",
[base, x])

    # use the function from the 'math' library
    return log(x, base)

def func_abs(x):
    """Return the absolute value of x"""

    # use the built-in function
    return abs(x)

def prime_factors(x):
    """Return a list of all of x's prime factors"""

    # invalid cases
    if x % 1 != 0 or x < 1:
        raise CalcOperationError("Must be a positive whole number", "LCM or HCF", [x])

    # typical cases
    factors = []
    i = 2

    # for each possible factor
    while i ** 2 <= x:

        # if it is a factor, add it and adjust x to prevent repeats
        # don't increment i as more than 1 of the same factor is possible
        if x % i == 0:
            x //= i
            factors.append(i)

        # if it's not a factor, increment it to find the next
        else:
            i += 1

    # add the last factor if x isn't 1
    if x > 1:
        factors.append(x)

    return factors

def to_dict(factors):
    """Gets a dictionary of all numbers and the number of times they occur"""

    factors_dict = {}
    for factor in factors:

        # if that prime factor is already in the dictionary, increment the count
        if factor in factors_dict:
            factors_dict[factor] += 1

        # otherwise, add it with an initial count of 1
        else:
            factors_dict[factor] = 1

    return factors_dict

def product_dict(dictionary):

    # the number of times each key appears is the value corresponding to that key
    # so the product is the product of all keys each to the power of their value
    product = 1
    for key in dictionary:
        product *= key ** dictionary[key]
```

```python
    return product

def func_lcm(x, y):
    """Return the lowest common multiple of x and y"""

    # gets dictionaries for each input with the prime factors
    # as the key and the number of occurances as the value
    prime_factors_x = to_dict(prime_factors(x))
    prime_factors_y = to_dict(prime_factors(y))

    # the prime factors of the LCM of x and y starts as those of x
    prime_factors_lcm = prime_factors_x.copy()

    for factor in prime_factors_y:

        # if the factor is not in LCM or it is in LCM but y has more, add the number y has
        if factor not in prime_factors_lcm or prime_factors_y[factor] > prime_factors_lcm[factor]:
            prime_factors_lcm[factor] = prime_factors_y[factor]

    # multiply all the prime factors together to get the LCM
    return product_dict(prime_factors_lcm)

def func_hcf(x, y):
    """Return the highest common factor of x and y"""

    # gets dictionaries for each input with the prime factors
    # as the key and the number of occurances as the value
    prime_factors_x = to_dict(prime_factors(x))
    prime_factors_y = to_dict(prime_factors(y))

    prime_factors_hcf = {}

    # the prime factors of the HCF are the minimum number that are in both x and y
    for factor in prime_factors_x:
        if factor in prime_factors_y:
            prime_factors_hcf[factor] = min(prime_factors_x[factor], prime_factors_y[factor])

    # multiply all the prime factors together to get the HCF
    return product_dict(prime_factors_hcf)

def func_quadp(a, b, c):
    """Return the positive square root answer of the quadratic equation ax^2 + bx + c = 0"""

    from Datatypes import Num

    discriminant = b**2 - 4*a*c

    # invalid cases
    if discriminant < 0:
        raise CalcOperationError("No real solutions", "quadp", [a, b, c])

    # quadratic formula with positive square root
    return (-b + discriminant**Num("0.5")) / (2 * a)

def func_quadn(a, b, c):
    """Return the positive square root answer of the quadratic equation ax^2 + bx + c = 0"""

    from Datatypes import Num

    discriminant = b**2 - 4*a*c

    # invalid cases
    if discriminant < 0:
        raise CalcOperationError("No real solutions", "quadn", [a, b, c])

    # quadratic formula with negative square root
    return (-b - discriminant**Num("0.5")) / (2 * a)

def func_rand(low, high):
    """Return a random integer between 'low' and 'high'"""

    # invalid cases
    if low % 1 != 0 or high % 1 != 0:
        raise CalcOperationError("Must be whole numbers", "rand", [low, high])

    # if the wrong way around, swap them
    if low > high:
```

```
        low, high = high, low

    # use the function from the 'random' library
    return randint(low, high)

def func_sin(x):
    """Return sin(x) where x is in radians"""

    # use the function from the 'math' library
    return sin(x)

def func_cos(x):
    """Return cos(x) where x is in radians"""

    # use the function from the 'math' library
    return cos(x)

def func_tan(x):
    """Return tan(x) where x is in radians"""

    from Datatypes import valid_tokens, Num

    # invalid cases
    if x % valid_tokens["pi"] == Num("0.5") * valid_tokens["pi"]:
        raise CalcOperationError("Tangent is undefined for values half way between multiples of pi", "tan",
[x])

    # use the function from the 'math' library
    return tan(x)

def func_arsin(x):
    """Return arsin(x) where the answer is in radians"""

    # invalid cases
    if x < -1 or x > 1:
        raise CalcOperationError("Inverse sine is only defined for values between -1 and 1 inclusive",
"arsin", [x])

    # use the function from the 'math' library
    return asin(x)

def func_arcos(x):
    """Return arcos(x) where the answer is in radians"""

    # invalid cases
    if x < -1 or x > 1:
        raise CalcOperationError("Inverse cosine is only defined for values between -1 and 1 inclusive",
"arcos", [x])

    # use the function from the 'math' library
    return acos(x)

def func_artan(x):
    """Return artan(x) where the answer is in radian"""

    # use the function from the 'math' library
    return atan(x)

def func_sinh(x):
    """Return sinh(x) where x is in radians"""

    # use the function from the 'math' library
    return sinh(x)

def func_cosh(x):
    """Return cosh(x) where x is in radians"""

    # use the function from the 'math' library
    return cosh(x)

def func_tanh(x):
    """Return tanh(x) where x is in radians"""

    # use the function from the 'math' library
    return tanh(x)

def func_arsinh(x):
    """Return arsinh(x) where the answer is in radians"""
```

```
    # use the function from the 'math' library
    return asinh(x)

def func_arcosh(x):
    """Return arcosh(x) where the answer is in radians"""

    # invalid cases
    if x < 1:
        raise CalcOperationError("Inverse hyperbolic cosine is undefined for values less than 1", "arcosh",
[x])

    # use the function from the 'math' library
    return acosh(x)

def func_artanh(x):
    """Return artanh(x) where the answer is in radian"""

    # invalid cases
    if x <= -1 or x >= 1:
        raise CalcOperationError("Inverse hyperbolic tangent is only defined for values between -1 and 1
exclusive", "artanh", [x])

    # use the function from the 'math' library
    return atanh(x)
```

PygameTools.py

```
"""
Tools for writing pygame windows
"""

import pygame as pg

COLOURS = {
    "black":    (0,    0,      0),
    "white":    (255,  255,    255),
    "grey":     (200,  200,    200),
    "red":      (255,  0,      0),
    "green":    (0,    200,    0),
    "blue":     (0,    0,      255),
    "yellow":   (255,  255,    0)
}

def format_text(text, MAX_CHARS_PER_LINE, MAX_LINES=None, break_anywhere=False):
    """Format text into lines so they fit on the screen and if it exceeds the maximum number of lines, trail
off..."""

    # if we allow breaking the string anywhere, we just treat the string as an array of characters
    if break_anywhere:
        words = text

    # otherwise we split the text into an array of words (however we haven't dealt with newlines yet)
    else:
        words = text.split(" ")

    # initialise the new 'lines' list and 'current_line' string as empty
    lines = []
    current_line = ""

    # for each word in the instructions:
    word_num = 0
    while (MAX_LINES is None or len(lines) < MAX_LINES) and len(words) > word_num:
        word = words[word_num]

        # while there is a newline character in the current word:
        while "\n" in word:

            # the line is the word before the newline character
            line = word.split("\n")[0]

            # if it can't fit on the current line, add the current line and it as 2 separate lines
            if len(line) + len(current_line) > MAX_CHARS_PER_LINE:
                lines.append(current_line)
                lines.append(line)

            # if it can fit, add the word (add a space between if we don't allow breaking words anywhere)
            else:
                if break_anywhere:
```

```
                    lines.append(current_line + line)
                else:
                    lines.append(current_line + " " + line)

            # the current line is empty as after a newline character we always start a new line when there is
a newline character
            current_line = ""

            # then replace the 'word' variable with all characters after the first newline character and
continue the loop
            word = "\n".join(word.split("\n")[1:])

        # if there is enough room to put the word on the current line, add it
        # otherwise, add it to a new line
        if len(word) + len(current_line) > MAX_CHARS_PER_LINE:
            lines.append(current_line)
            current_line = word
        else:
            # only add a space if we don't allow breaking anywhere
            if break_anywhere:
                current_line += word
            else:
                current_line += " " + word

        word_num += 1

    # if we have run out of room, trail off
    if MAX_LINES is not None and len(lines) == MAX_LINES:
        lines[-1] += "..."

    # otherwise, add the last line
    else:
        lines.append(current_line)

    return lines

def middle_box(box):
    """Return the middle position of 'box'"""
    return box[0] + box[2] // 2, box[1] + box[3] // 2

class Text:
    """Text object facilitating drawing to the screen and changing the text message"""
    def __init__(self, text_message, size, colour, center_pos, font, display):
        self.__font = pg.font.Font(font, round(size))
        self.__colour = colour
        self.__center_pos = center_pos
        self.__display = display
        self.__surf = self.__font.render(str(text_message), True, colour)
        self.__rect = self.__surf.get_rect()
        self.__rect.center = center_pos
    def draw(self):
        """Draw the text to the display"""
        self.__display.blit(self.__surf, self.__rect)
    def edit_text_message(self, new_text_message):
        """Edit the text message"""
        self.__surf = self.__font.render(str(new_text_message), True, self.__colour)
        self.__rect = self.__surf.get_rect()
        self.__rect.center = self.__center_pos

class Button:
    """Button object facilitating drawing it and the text in it to the screen, checking whether a point lies
within it and changing the text message on the box"""
    def __init__(self, box, colour, text, display):
        self.__box = box
        self.__colour = colour
        self.__text = text
        self.__display = display
    def draw(self):
        """Draw the button and text in it to the display"""
        pg.draw.rect(self.__display, self.__colour, self.__box)
        self.__text.draw()
    def is_within(self, mouse_pos):
        """Return whether or not the mouse is within the button"""
        return self.__box[0] <= mouse_pos[0] <= self.__box[0] + self.__box[2] and self.__box[1] <=
mouse_pos[1] <= self.__box[1] + self.__box[3]
    def edit_text_message(self, new_text_message):
        """Edit the text message on the button"""
        self.__text.edit_text_message(new_text_message)
```

```
class Draw:
    """Drawer object which facilitates creating buttons and text in pygame"""
    def __init__(self, display, font):
        self.__display = display
        self.__font = font
    def button(self, box, box_colour, text_message, text_size, text_colour):
        """Create a button"""
        return Button(box, box_colour, self.text(text_message, text_size, text_colour, middle_box(box)),
self.__display)
    def text(self, text_message, size, colour, center_pos):
        """Create text"""
        return Text(text_message, size, colour, center_pos, self.__font, self.__display)
```

README.md

```
# Calculator

A scientific calculator on the computer

## Users
Users can run the following files:
* __'UserInterface.pyw'__ for a graphical user interface
* __'Interface.py'__ for a command-line interface with memory
* __'Calc.py'__ for a command-line interface without memory

## Programmers

### To calculate the answer to a single expression
Use the __'calculate'__ function from the file __'Calc.py'__

### To create a custom user interface using my memory system
Instantiate the __'Interface'__ class in the file __'Interface.py'__ and:
* if the user wants to calculate the answer to an expression, use the __'calculate'__ method
* if the user wants to view instructions, use the __'instructions'__ attribute
* if the user wants to view memory, use the __'recent_memory'__ method
* if the user wants to clear memory, use the __'clear_memory'__ method
* if the user wants to insert a specific memory answer into their expression:
   1. get which memory item is being requested
   1. use the __'memory_item'__ method to get the original expression and answer of interest
   1. it may be best to re-calculate the answer using the original expression and the __'calculate'__ method
   1. insert the answer from memory or the re-calculated answer into the expression

NOTE: before calling the __'recent_memory'__ or __'memory_item'__ methods with a number from the user, the interface should call __'len_memory'__ to check how many items are in memory and verify the number wanted is a valid number and equal to or less than the number of items in memory. If not, display the relevant error message. If either of these methods are called with invalid parameters, they will raise __'IndexError'__.

### To create a custom user interface without my memory system
1. use the __'calculate'__ function in __'Calc.py'__ to call the calculator with an expression
1. catch any errors derived from __'CalcError'__ in __'Errors.py'__ and present the message to the user
1. add to and present to the user the instructions from the global variable __'instructions'__ in __'Calc.py'__

### To add custom operations to the calculator
1. write a function to execute the operation in __'Operations.py'__
1. import this into __'Datatypes.py'__
1. add details of the operation including the function to the __'valid_tokens'__ dictionary at the bottom of __'Datatypes.py'__
1. explain how to use it in __'Instructions.txt'__

## Instructions

Enter an expression to calculate the answer.

Operators are represented by a symbol and perform an operation on the numbers around them. Binary operators have 2 numbers (1 either side of the symbol), whereas unary operators have 1 number (either left or right of the symbol). Functions are represented by a word followed by brackets containing all operands (values needed for the function) separated by commas. Constants are a word representing a number very accurately. Simply enter the word and it will convert it to the number.
```

Functions and constants are always executed first and then operators are executed using BODMAS - brackets, other (exponents and unary operators), division and multiplication, addition and subtraction.

Binary Operators
* Addition: use '+' between 2 numbers to find the first add the second.
* Subtraction: use '-' between 2 numbers to find the first subtract the second.
* Multiplication: use '*' between 2 numbers to find the first multiplied by the second.
* Division (true): use '/' between 2 numbers to find the first divided by the second.
* Division (floor): use '\\' between 2 numbers to find the first divided by the second and rounded down to the nearest whole number.
* Mod: use '%' between 2 numbers to find the remainder after the first is divided by the second.
* Exponentiation: use '^' between 2 numbers to find the first to the power of the second.
* Root: use '¬' to find the first th root of the second - '2¬a' is the square root of 'a', '3¬a' is the cube root of 'a', etc.
* Permutations: use 'P' to find the number of ways there are to organise the second number of items into the first number of places including all possible orders.
* Combinations: use 'C' to find the number of ways there are to organise the second number of items into the first number of places only counting 1 possible order.

Unary Operators
* Positive: use '+' before a number to find the positive of it.
* Negative: use '-' before a number to find the negative of it.
* Factorial: use '!' after a positive whole number to find the product of all positive whole numbers less than or equal to it.

Functions
* Natural log: use 'ln' with 1 operand to find the natural logarithm of it.
* Logarithm: use 'log' with 2 operands to find the logarithm of the first to the second base.
* Absolute value: use 'abs' with 1 operand to find the absolute value of it which is always positive.
* Lowest common multiple: use 'lcm' with 2 operands to find the lowest common multiple of them.
* Highest common factor: use 'hcf' with 2 operands to find the highest common factor of them.
* Random number generator: use 'rand' with 2 operands to find a random integer between them, inclusive.
* Quadratic equation solver: use 'quadp' with 3 operands (a, b and c) to find the positive square root answer to the quadratic equation 'ax^2 + bx + c = 0' or use 'quadn' to find the negative square root answer of the same equation.
* Sine: use 'sin' with 1 operand (an angle) to find the ratio between the opposite side and hypotenuse of its triangle.
* Cosine: use 'cos' with 1 operand (an angle) to find the ratio between the adjacent side and hypotenuse of its triangle.
* Tangent: use 'tan' with 1 operand (an angle) to find the radio between the opposite and adjacent sides of its triangle.
* Inverse sine: use 'arsin' with 1 operand between -1 and 1 inclusive to find the angle it makes with the opposite side and hypotenuse of its triangle.
* Inverse cosine: use 'arcos' with 1 operand between -1 and 1 inclusive to find the angle it makes with the adjacent side and hypotenuse of its triangle.
* Inverse tangent: use 'artan' with 1 operand to find the angle it makes with the opposite and adjacent sides of its triangle.
* Hyperbolic sine: use 'sinh' with 1 operand (an angle) to find the ratio between the opposite side and hypotenuse of its hyperbola.
* Hyperbolic cosine: use 'cosh' with 1 operand (an angle) to find the ratio between the adjacent side and hypotenuse of its hyperbola.
* Hyperbolic tangent: use 'tanh' with 1 operand (an angle) to find the ratio between the opposite and adjacent sides of its hyperbola.
* Inverse hyperbolic sine: use 'arsinh' with 1 operand to find the angle it makes with the opposite side and hypotenuse of its hyperbola.
* Inverse hyperbolic cosine: use 'arcosh' with 1 operand at least 1 to find the angle it makes with the adjacent side and hypotenuse of its hyperbola.
* Inverse hyperbolic tangent: use 'artanh' with 1 operand between -1 and 1 inclusive to find the angle it makes with the opposite and adjacent sides of its hyperbola.

Constants
* pi: use 'pi' to get the ratio between a circle's circumference and its diameter. Value = 3.1415...
* tau: use 'tau' to get 2 lots of pi - the number of radians in 360 degrees. Value = 6.2831...
* e: use 'e' to get Euler's number. Value = 2.7182...
* g: use 'g' to get the acceleration due to gravity close to the Earth's surface. Value = 9.80665
* phi: use 'phi' to get the golden ratio found in many places in nature. Value = 1.6180...

UserInterface.pyw

```
"""
A graphical user interface for the calculator
To open, instantiate the 'Window' class and then call the 'run' method
"""

import pygame as pg
from PygameTools import COLOURS, Draw, format_text
from Interface import Interface
from Errors import CalcError

class Window:
```

```python
    def __init__(self):

        # constants
        self.__RESOLUTION = self.__WIDTH, self.__HEIGHT = 800, 600
        self.__TARGET_FPS = 30
        self.__BACKGROUND_COLOUR = COLOURS["grey"]
        self.__FONT = "freesansbold.ttf"

        # variables needed for the calculator
        self.__calculator = Interface()
        self.__expr = self.__ans = self.__error_msg = ""

        # the current mode name and mapping between mode names and their methods
        self.__mode = "normal"
        self.__modes = {
            "normal": self.__mode_normal,
            "instructions": self.__mode_instructions
        }

        # the distance the instructions scrollable view is positioned above the top of the screen
        self.__scroll = 0

        # whether or not to exit the calculator
        self.__done = False

    def run(self):
        """Run the calculator user interface"""

        # start pygame, create the window, caption it and start the clock
        pg.init()
        self.__display = pg.display.set_mode(self.__RESOLUTION)
        pg.display.set_caption("Calculator")
        clock = pg.time.Clock()

        # create my drawer for drawing things on the screen
        self.__drawer = Draw(self.__display, self.__FONT)

        # create buttons and text I will draw later
        self.__button_instructions = self.__drawer.button((0, 0, 200, 50), COLOURS["yellow"], "Instructions",
25, COLOURS["black"])
        self.__button_clear_memory = self.__drawer.button((200, 0, 200, 50), COLOURS["red"], "Clear Memory",
25, COLOURS["black"])
        self.__button_back = self.__drawer.button((self.__WIDTH - 100, 0, 100, 50), COLOURS["black"], "Back",
25, COLOURS["white"])
        self.__texts_expr = []
        self.__texts_ans = []
        self.__texts_error_msg = []
        self.__text_instructions_title = self.__drawer.text("Instructions", 25, COLOURS["yellow"], (400, 50))
        self.__text_memory = self.__drawer.text("Memory", 25, COLOURS["green"], (700, 50))
        self.__buttons_memory = []
        self.__text_extra_memory = []
        self.__format_instructions()

        # main loop
        while not self.__done:

            # clear screen
            self.__display.fill(self.__BACKGROUND_COLOUR)

            # get events
            events = pg.event.get()

            # let windows close the window
            for event in events:
                if event.type == pg.QUIT:
                    self.__done = True

            if not self.__done:

                # call the current mode's method
                self.__modes[self.__mode](events)

                # update the display and tick the clock
                pg.display.update()
                clock.tick(self.__TARGET_FPS)

        # close the pygame window
```

```
        pg.quit()

    def __mode_normal(self, events):
        """Runs every tick when in normal mode"""

        # draw buttons and text
        self.__button_instructions.draw()
        self.__button_clear_memory.draw()
        self.__text_memory.draw()
        for button in self.__buttons_memory:
            button.draw()
        for text in self.__texts_expr + self.__texts_ans + self.__texts_error_msg + self.__text_extra_memory:
            text.draw()

        # draw lines between the buttons to distinguish them (same colour as background so when they aren't
there it looks the same)
        for count in range(2, 5+1):
            pg.draw.line(self.__display, self.__BACKGROUND_COLOUR, (600, 100 * count), (800, 100 * count))

        # handle events
        for event in events:
            if event.type == pg.KEYDOWN:

                # if the user pressed escape, clear the expression
                if event.key == pg.K_ESCAPE:
                    self.__expr = ""
                    self.__update_text_and_buttons(expr=True)

                # if the user presses backspace, remove 1 character from the expression
                elif event.key == pg.K_BACKSPACE:
                    self.__expr = self.__expr[:-1]
                    self.__update_text_and_buttons(expr=True)

                # if either or the return/enter keys are pressed, call the 'calculate' method
                elif event.key == pg.K_RETURN or event.key == pg.K_KP_ENTER:
                    self.__calculate()

                # otherwise add the text to the expression
                else:

                    # if the first thing they type isn't a number, insert the last answer into the start of
the expression
                    if event.key not in [pg.K_0, pg.K_1, pg.K_2, pg.K_3, pg.K_4, pg.K_5, pg.K_6, pg.K_7,
pg.K_8, pg.K_9, pg.K_KP0, pg.K_KP1, pg.K_KP2, pg.K_KP3, pg.K_KP4, pg.K_KP5, pg.K_KP6, pg.K_KP7, pg.K_KP8,
pg.K_KP9] and self.__expr == "":
                        self.__expr = self.__ans

                    self.__ans = self.__error_msg = ""
                    self.__expr += event.unicode
                    self.__update_text_and_buttons(expr=True, ans=True, error=True)

            elif event.type == pg.MOUSEBUTTONDOWN and event.button == 1:
                mouse_pos = pg.mouse.get_pos()

                # if the user clicked the instructions button, change to instructions mode
                if self.__button_instructions.is_within(mouse_pos):
                    self.__mode = "instructions"

                # if the user clicked the clear memory button, clear the memory
                elif self.__button_clear_memory.is_within(mouse_pos):
                    self.__calculator.clear_memory()
                    self.__ans = self.__error_msg = ""
                    self.__update_text_and_buttons(memory=True, ans=True, error=True)

                # if the user clicked on a memory item, insert that item into the expression
                else:
                    for button in self.__buttons_memory:
                        if button.is_within(mouse_pos):
                            self.__expr += "M{}".format(self.__buttons_memory.index(button) + 1)
                            self.__update_text_and_buttons(expr=True)

    def __mode_instructions(self, events):
        """Runs every tick when in instructions mode"""

        # draw the instructions text onto the intemediate surface
        for line in self.__texts_instructions:
            line.draw()
```

```
        # draw the intemediate surface onto the main surface
        self.__display.blit(self.__intermediate, (0, 100 + self.__scroll))

        # draw a rectangle, the title and back button over the top of the top of the surface
        pg.draw.rect(self.__display, self.__BACKGROUND_COLOUR, (0, 0, self.__WIDTH, 100))
        self.__text_instructions_title.draw()
        self.__button_back.draw()

        # handle events
        for event in events:

            # if the user pressed escape or clicked the back button, change to normal mode
            if event.type == pg.KEYDOWN:
                if event.key == pg.K_ESCAPE:
                    self.__mode = "normal"
            elif event.type == pg.MOUSEBUTTONDOWN:
                if event.button == 1:
                    mouse_pos = pg.mouse.get_pos()
                    if self.__button_back.is_within(mouse_pos):
                        self.__mode = "normal"

                # if the user scrolled up, lower the intemediate surface but not lower than the origin
                elif event.button == 4:
                    self.__scroll = min(self.__scroll + 15, 0)

                # if the user scrolled down, raise the intemediate surface but not higher than the point
                # where the bottom instructions line is fully visible
                elif event.button == 5:
                    self.__scroll = max(self.__scroll - 15, self.__HEIGHT - len(self.__texts_instructions) *
20 - 100)

    def __clean_up_expr(self, expr):
        """Remove whitespace and extra brackets on the outside of the expression"""

        expr = expr.strip()
        if len(expr) > 0:
            while expr[0] == "(" and expr[-1] == ")":
                expr = expr[1:-1].strip()

        return expr

    def __calculate(self):
        """Calculate the answer to the expression and update everything"""

        # remove whitespace at the start and end, make lower case and replace 'ans' with 'm1'
        expr = self.__expr.strip().lower().replace("ans", "m1")

        # replace all memory references with the actual answers, clean up the expression and call
        # the main calculator with the resulting expression, catching errors and displaying them
        try:
            self.__ans =
self.__calculator.calculate(self.__clean_up_expr(self.__convert_memory_references(expr)))
        except CalcError as e:
            self.__error_msg = str(e)
            self.__ans = ""
            self.__update_text_and_buttons(ans=True, error=True)
        else:
            self.__error_msg = ""
            self.__expr = ""
            self.__update_text_and_buttons(True, True, True, True)

    def __convert_memory_references(self, expr):
        """Convert all memory references to the actual answers"""

        # runs for every 'm' in 'expr'
        for _ in range(expr.count("m")):

            # get the string after the first 'm'
            index = expr.split("m")[1]

            # find the number of characters until the first non-numerical character
            pos = 0
            while pos < len(index) and index[pos] in "0123456789":
                pos += 1

            # find the string between the first 'm' and the first non-numerical characters
            # the slice means all characters up to pos characters through it
            index = index[:pos]
```

```
            # if there is a number after the 'm':
            if index != "":
                index = int(index)

                # if valid, replace the reference with the actual answer
                if 1 <= index <= self.__calculator.len_memory():
                    expr = expr.replace("m{}".format(index), "(" + self.__calculator.memory_item(index)[1] +
")", 1)

                # otherwise, give an error message
                else:
                    raise CalcError("Memory references must be between 1 and the number of items in memory")

        return expr

    def __update_text_and_buttons(self, memory=False, expr=False, ans=False, error=False):
        """
        Update the message on text and button objects if the message has changed
        The parameters are whether or not to update the message on those text/button objects
        """

        # if we need to update the memory buttons:
        if memory:

            self.__text_extra_memory = []
            count = 0

            # for the most recent 5 items in memory:
            for expression, answer in self.__calculator.recent_memory(5):

                # format the text for each memory item into lines
                lines = format_text("{}: {} ({})".format(count + 1, answer, expression), 15, 3, True)

                # if there is only 1 line, make it
                if len(lines) == 1:

                    # if there is already a button, change the text
                    if len(self.__buttons_memory) > count:
                        self.__buttons_memory[count].edit_text_message(lines[0])

                    # otherwise create a button with the new text
                    else:
                        self.__buttons_memory.append(self.__drawer.button((600, 100 * (count + 1), 200, 100),
COLOURS["black"], lines[0], 20, COLOURS["white"]))

                # otherwise make it and add the other 1 or 2 lines
                else:
                    # if there is already a button, change the text
                    if len(self.__buttons_memory) > count:
                        self.__buttons_memory[count].edit_text_message(lines[1])

                    # otherwise create a button with the new text
                    else:
                        self.__buttons_memory.append(self.__drawer.button((600, 100 * (count + 1), 200, 100),
COLOURS["black"], lines[1], 20, COLOURS["white"]))

                    # add the first line
                    self.__text_extra_memory.append(self.__drawer.text(lines[0], 20, COLOURS["white"], (700,
(100 * count) + 125)))

                    # if there are 3 lines, add this too
                    if len(lines) == 3:
                        self.__text_extra_memory.append(self.__drawer.text(lines[2], 20, COLOURS["white"],
(700, (100 * count) + 175)))

                count += 1

            # remove buttons that aren't in memory anymore
            self.__buttons_memory = self.__buttons_memory[:count]

        # if we need to update the expression text objects:
        if expr:

            # format the expression into lines
            lines = format_text(self.__expr, 18, 5, True)
            count = 0
            for line in lines:
```

```
                # if there are already enough objects, change the message on it
                if len(self.__texts_expr) > count:
                    self.__texts_expr[count].edit_text_message(line)

                # otherwise, create a new object with the message
                else:
                    self.__texts_expr.append(self.__drawer.text(line, 35, COLOURS["blue"], (300, 200 + (30 *
count))))

                count += 1

            # remove unnecessary objects
            self.__texts_expr = self.__texts_expr[:count]

        # if we need to update the answer text objects:
        if ans:

            # format the answer into lines
            lines = format_text(self.__ans, 18, 5, True)
            count = 0
            for line in lines:

                # if there are already enough objects, change the message on it
                if len(self.__texts_ans) > count:
                    self.__texts_ans[count].edit_text_message(line)

                # otherwise, create a new object with the message
                else:
                    self.__texts_ans.append(self.__drawer.text(line, 35, COLOURS["green"], (300, 400 + (30 *
count))))

                count += 1

            # remove unnecessary objects
            self.__texts_ans = self.__texts_ans[:count]

        # if we need to update the error text objects:
        if error:

            # format the error message into lines
            lines = format_text(self.__error_msg, 40, 3)
            count = 0
            for line in lines:

                # if there are already enough objects, change the message on it
                if len(self.__texts_error_msg) > count:
                    self.__texts_error_msg[count].edit_text_message(line)

                # otherwise, create a new object with the message
                else:
                    self.__texts_error_msg.append(self.__drawer.text(line, 25, COLOURS["red"], (300, 100 +
(25 * count))))

                count += 1

            # remove unnecessary objects
            self.__texts_error_msg = self.__texts_error_msg[:count]

    def __format_instructions(self):
        """Format the instructions into lines on a scrollable surface"""

        # extend the instructions
        instructions = "SCROLL DOWN TO VIEW MORE:\n\n" + self.__calculator.instructions + "\n\nTo insert a
previous answer into the expression, click on the item in the memory section. You can also type 'ans' to
insert the last answer into the expression or 'Mx' to insert the xth answer into the expression.\n\nYou can
press ESCAPE at any time to clear the expression and when you start typing on an empty expression it will add
the previous answer before it unless you type a number of just pressed ESCAPE."

        # format the instructions into lines
        lines = format_text(instructions, 75)

        # make the intemediate surface just bit enough to hold all the instruction lines and make it the
background colour
        self.__intermediate = pg.surface.Surface((self.__WIDTH, len(lines) * 20 + 10))
        self.__intermediate.fill(self.__BACKGROUND_COLOUR)

        # make a new drawer to draw on the intemediate surface
```

```
        new_drawer = Draw(self.__intermediate, self.__FONT)

        # create a text object for each line
        self.__texts_instructions = []
        count = 0
        for line in lines:
            self.__texts_instructions.append(new_drawer.text(line, 20, COLOURS["black"], (400, 10 + 20 * 
count)))
            count += 1

if __name__ == "__main__":
    Window().run()
```