2022/2023

Practical 10 - Final Assignment

For your final assignment you should complete all the tasks on this worksheet. Each task asks you to write a function; in your code you should name your functions appropriately, and provide a docstring, so that it is clear which part of the assignment each function corresponds to. You should submit all of your work in a single `.py` file, which when run, runs the programme described at the end of the final task. You should also submit a text file explaining in as much detail as you think is appropriate how each of your functions operates; again, between 50 and 200 words will be appropriate for each part of the assignment, but if you feel you need to add more detail you are welcome to do so.

Submit by 12midday, Monday 16th January 2023.

Handling mathematical expressions

In this task we are going to build up a programme capable of handling mathematical expressions involving numbers, letter variables and named functions.

Part 1

Write a function which takes a string as input, and checks whether any brackets occurring in the string appear as correctly matched pairs of one opening bracket `(` and one closing bracket `)`. That is, the brackets in the following four strings are all correctly matched: `"()"`; `"()()"`; `"(())"` `"((()))"`; . However, in the following strings, the brackets are not correctly matched: `"("`; `")("`; `"())"`; `"())(()"`. Your programme should output `True` if all brackets in the string are correctly matched and `False` otherwise. We are only considering ordinary round brackets, not any other kind of bracket.

Hint: If a string contains mismatched brackets, we must have either opened a pair of brackets which we haven't subsequently closed, or (looking from the start of the string to the end) tried to close more pairs of brackets than we have opened.

Test inputs:

Input	Expected output
`"()"`	`True`
`"a"`	`True`
`"(b())"`	`True`
`"("`	`False`
`"()a)"`	`False`
`"|)"`	`False`
`"(]"`	`False`
`"([)"`	`True`
`"[["`	`True`
`"(a(bb)ccc)dddd"`	`True`
`"(a)bb()ccc"`	`True`

Part 2a:

Write a function which takes a string as input and returns the contents of the first matched pair of brackets which appear in the string. For example, given the input `"Some text (in brackets)"` your function should return the string `"in brackets"`. You should use the function you wrote for Part 1 to check whether the input string contains mismatched brackets; if it does, print an error message and `return None`.

Input	Expected output
`"(a)"`	`"a"`
`"()"`	`""`
`"abcd"`	`""`
`"a (b) c (d)"`	`"b"`
`"((a) b (c) d)"`	`"(a) b (c) d"`
`"(a(bb)ccc)dddd"`	`"a(bb)ccc"`
`"()a)"`	`None`

Part 2b:

Write a separate function (perhaps starting with the code from your function in Part 2a) which takes a string as input and returns the contents of the first pair of brackets, except not including anything contained within any further nested brackets. For example, given the input `"(a(bb)ccc)dddd"` your function should return `"accc"`, and given the input `"((a) b (c) d)"` your function should return `" b d"`. Again use the function you wrote for Part 1 to check whether the input string contains mismatched brackets; if it does, print an error message and `return None`.

Input	Expected output
`"(a)"`	`"a"`
`"((a))"`	`""`
`"((a)b)"`	`"b"`
`"a (b (c) d) e"`	`"b  d"`
`"(a(bb)ccc)dddd"`	`"accc"`
`"((a) b (c) d)"`	`" b  d"`
`"((a)b"`	`None`

Part 3:

Mathematical expressions and well-formed terms

A mathematical expression consists of various *terms* connected by mathematical *operators*, for example the expression $2x + y$ has terms $2x$ and y and the operator $+$. Of course the term $2x$ itself is actually an abbreviated shorthand for $2 \times x$, which contains the terms 2 and x and the operator $\times$.

We will define a well-formed term in the following way:

Rule	Examples
Any single letter (which we will interpret as a variable, in the mathematical sense) is a well-formed term	`x` is a well-formed term; `a` is a well-formed term;
Any natural number is a well-formed term	`3` is a well-formed term; `52837` is a well-formed term;
If s and t are well-formed terms, then:	
`(`s`+`t`)` is a well-formed term	`(x+3)` is a well-formed term, where s is the term `x` and t is the term `3`; `(52837+(x+3))` is a well-formed term, where s is the term `52837` and t is the term `(x+3)`;
`(`s`-`t`)` is a well-formed term	`(y-1)` is a well-formed term, where s is the term `y` and t is the term `1`; `((x-3)-y)` is a well-formed term, where s is the term `(x-3)` and t is the term `y`;
`(`s`*`t`)` is a well-formed term	`(2*(y-1))` is a well-formed term, where s is the term `2` and t is the term `(y-1)`; `((x-(2*x))*y)` is a well-formed term, where s is the term `(x-(2*x))` and t is the term `y`;
`(`s`/`t`)` is a well-formed term	`(a/b)` is a well-formed term, where s is the term `a` and t is the term `b`; `(1/(x+1))` is a well-formed term, where s is the term `1` and t is the term `(x+1)`;
Rule	Examples
If s is a well-formed term and n is a natural number greater than or equal to 1, then:	
`(`s`^`n`)` is a well-formed term	`(x^2)` is a well-formed term; `((y*(x-(2*x)))^52837)` is a well-formed term.

Note that we can apply our formation rules in any order. For example, `(((x^2)*(x/(y^3)))+1)` is a well-formed term.

Notice that on the left and right of any operator, we either have a single letter variable, a natural number, or a term enclosed in a pair of brackets. Note also that every individual operator corresponds to a specific left bracket and right bracket. For example, the `+` operator in the last example corresponds to the outside brackets: `(((x^2)*(x/(y^3)))+1)`, and the `*` operator corresponds to the brackets indicated in blue: `(((x^2)*(x/(y^3)))+1)`

Task:

Write a function which takes a well-formed expression (as a Python string) as input and returns a *parse tree* of that expression as a *nested list*, according to the following specification:
Parse tree of **input:**
If the input represents a single-letter variable or a natural number:
Output the input
Else the input is a term enclosed in brackets, containing an operator correponding to the outermost

pair of brackets, and this operator appears between a left-hand term and a right-hand term (which may be single-letter variables, natural numbers, or themselves terms enclosed in brackets):
Output a list `[operator, left-hand tree, right-hand tree]`, where `operator` is a single-letter Python string containing the operator that corresponds to the outermost pair of brackets in the input, `left-hand tree` is the parse tree for the term which appears to the left of the operator, and `right-hand tree` is the parse tree for the term which appears to the right of the operator.

Test inputs:

Input	Expected output
`"52837"`	`"52837"`
`"x"`	`"x"`
`"(a+b)"`	`['+', "a", "b"]`
`"((a+b)^2)"`	`['^', ['+', "a", "b"], "2"]`
`"((a+b)/(374-c))"`	`['/', ['+', "a", "b"], ['-', "374", "c"]]`
`"(((x^2)*(x/(y^3)))+1)"`	`['+', ['*', ['^', "x", "2"], ['/', "x", ['^', "y", "3"]]], "1"]`

Part 4:

We are going to extend our definition of a well-formed term to allow us to consider further mathematical functions, by including the following rules: (remember, a well-formed term is just a Python string of text, it doesn't include any actual mathematical functionality)

Rule	Examples
If t is a well-formed term, then:	
`exp(`t`)` is a well-formed term	`exp(x)` is a well-formed term;
`log(`t`)` is a well-formed term	`log((a+b))` is a well-formed term;
`sin(`t`)` is a well-formed term	`(sin(1)/y)` is a well-formed term;
`cos(`t`)` is a well-formed term	`cos(exp((x/2)))` is a well-formed term;

Now, we have four distinct kinds of terms. From before, we have terms which are either a single letter variable, a natural number, or a term enclosed in a pair of brackets corresponding to an operator. We now have a fourth kind of term, consisting of a function name followed by a corresponding pair of brackets.

Task:

Extend your function from Part 3 to meet the following specification:
Parse tree of **input:**
If the input represents a single-letter variable or a natural number:
Output the input
Else if the input is a function name followed by an opening bracket, with the final character of the input being a closing bracket: **Output** a list `[function_name, argument tree]`, where `function` is the appropriate one of the four possible Python strings `exp`, `log`, `sin` or `cos`; and `argument tree` is the parse tree for the term which appears between the inital opening bracket and the final closing bracket
Else the input is a term enclosed in brackets, containing an operator correponding to the outermost pair of brackets, and this operator appears between a left-hand term and a right-hand term (which may be single-letter variables, natural numbers, or themselves terms enclosed in brackets):
Output a list `[operator, left-hand tree, right-hand tree]`, where `operator` is a single-

letter Python string containing the operator that corresponds to the outermost pair of brackets in the input, `left-hand tree` is the parse tree for the term which appears to the left of the operator, and `right-hand tree` is the parse tree for the term which appears to the right of the operator.

Test inputs:

Input	Expected output
`"exp(1)"`	`["exp", "1"]`
`"(a+sin(x))"`	`['+', "a", ["sin", "x"]]`
`"(log(r)^3)"`	`['^', ["log", "r"], "3"]`
`"log(log(((2*n)+1)))"`	`["log", ["log", ['+', ['*', "2", "n"], "1"]]]`

Part 5: Symbolic Differentiation

We will have learned at school how to differentiate mathematical functions with respect to a single variable. From this point forward, we will consider differentiation with respect to the variable x. This means that any other single-letter variable just represents a constant coefficient. We have the following well-known derivatives:

Function ($f(x)$)	Derivative with respect to x $\left(\frac{df}{dx}\right)$
Any constant (including natural numbers and variables)	0
x	1
x^n where n is a natural number ≥ 1	$n \times x^{n-1}$
$\exp(x)$	$\exp(x)$
$\log(x)$	$\frac{1}{x}$
$\sin(x)$	$\cos(x)$
$\cos(x)$	$-\sin(x)$

Furthermore, given functions f and g, we have the following well-known rules for the derivative of various combinations of f and g, in terms of their derivatives:

Function	Derivative with respect to x
$f + g$	$\frac{df}{dx} + \frac{dg}{dx}$
$f \cdot g$	$f \cdot \frac{dg}{dx} + g \cdot \frac{df}{dx}$
$f(g(x))$	$\frac{dg}{dx} \times \frac{df}{dx}(g(x))$
f/g	$\frac{g \cdot \frac{df}{dx} - f \cdot \frac{dg}{dx}}{g^2}$

Task:

Use these rules and derivatives to write a function which takes a well-formed term as input and outputs the term's derivative with respect to x, as a well-formed term. Here you should interpret `*` as multiplication, `^` as raising to a power (note we are not using the Python syntax for raising to a power), `exp()` as the exponential function, `log()` as the natural logarithm, and `sin()` and `cos()` as being the sine and cosine of an angle in radians.

Write a programme including your function which takes an input from the user and prints the derivative.

Test inputs:

Input	Expected output
`"32"`	`"0"`
`"a"`	`"0"`
`"x"`	`"1"`
`"(1+x)"`	Any well-formed term representing an expression equal to 1
`"(x^5)"`	`"(5*(x^4))"`
`"((x^3)-(x^2))"`	Any well-formed term representing an expression equal to $3x^2 - 2x$
`"exp(1)"`	Any well-formed term representing an expression equal to 0
`"exp(x)"`	Any well-formed term representing an expression equal to e^x
`"exp((x^2))"`	Any well-formed term representing an expression equal to $2xe^{x^2}$
`"((x^2)*exp(x))"`	Any well-formed term representing an expression equal to $2xe^x + x^2e^x$
`"log(sin(x))"`	Any well-formed term representing an expression equal to $\frac{\cos x}{\sin x}$
`"log(sin((x^2)))"`	Any well-formed term representing an expression equal to $\frac{2x\cos(x^2)}{\sin(x^2)}$