Lab 6

S2 2017

Lab 6

Objectives

The purpose of this week's lab is to:

- Review what you learned in the first half of the course.
- Practice programming and problem solving.

Remember: If you have any questions about or difficulties with any of the material covered in the course so far, now is the time to ask your tutor for help during the lab!

Exercise 0: Valid names

A name (of a variable, function, class, or anything) in python must consist only of letters (in the `a-z` range, lower or upper case), digits and underscores, and must not begin with a digit.

(a) Write a function `valid_name(string)` that takes as argument a string and determines if the string is a valid name. The function should return truth value (that is, a value of type `bool`).

(b) Actually, there is one more requirement on valid names: keywords in the python language cannot be used as names. You can find a list of python 3 keywords in Chapter 2 of Downey's book (section "Variable names"), in section 1.4.5 in Punch & Enbody's book, or in the online python documentation. Modify your function so that it checks that the argument string is not a keyword.

Exercise 1: Expressions, values and types

An *expression* is made up of constants (known as *literals* in a programming language), operators and function calls. Evaluating an expression results in a value, and every value has a type.

(a) Strictly speaking, operators in python are either unary (take one argument) or binary (take two arguments). Thus, an expression like `1 + 2 + 3` is really interpreted as `(1 + 2) + 3`. We can represent

this as a nested sequence, `( (1, '+', 2), '+', 3 )`. Note that each of the tuples in this sequence has three elements, where the middle one is the operator and the other two are the arguments. (The operator is written as a string, since otherwise it would generate a syntax error. In the functions below, you only need to be able to check which operator the string in the middle position represents.)

Download this file: expressions.py. In it is the definition of a function called `evaluate_expression(exp)` that takes as argument an expression in the form of a nested tuple, as shown above, and computes the value of the expression. The function handles only the standard arithmetic operators (`-`, `+`, `*`, `/`, `//` and `**`). Note that minus can appear as a unary operator, as in the expression `( ('-', 2), '*', 2 )`.

Try the function on the following expressions:

- `( 1, '-', ( 2, '-', 3 ) )`
- `( ( 1, '-', 2 ), '-', 3 )`
- `( (25, '//', 5), '%', 5 )`
- `( (25, '//', 5), '//', 5 )`
- `( 25, '//', (5, '//', 5) )`

Compare the value returned with what you get if you simply type the corresponding expression into the python shell. In which cases do you need parentheses to get the right grouping of subexpressions?

You can add some `print` calls to the function (for example, one at the top of the function) so that you can see the order in which subexpressions are evaluated.

(b) The type of the value returned by an expression is determined by the type of the arguments and the operator that is applied to them. For example, if `a` and `b` are integers, `a + b` is an integer, but `a / b` results in a float; if `a` and `b` are strings, then `a + b` is a string.

Write a function `type_of_expression(exp)` that takes an expression, in the form of a nested tuple as above, and returns the type of the value that results from evaluating it. To keep it simple, you can assume that all constants in the expression are either numbers (type `int` or `float`) or strings (type `str`) and that the operators are the standard arithmetic operators (as in part (a)). If the expression applies an operator to an argument of a type that is not correct for it, your function should print an error message and return `None`.

Hint: You can write this function following exactly the same structure as in the provided example function `evaluate_expression`. You only need to change what value is returned.

Test cases

- `( 2, '+', ( 4, '/', 2 ) )`
- `( '2', '*', ( 2, '+', 1 ) )`
- `( ( '2', '*', 2 ), '+', '1' )`
- `( ( '2', '*', '2' ), '+', '1' )`
- `( ( '2', '*', 2 ), '+', 1 )`

(c) (*advanced*) To write an unambiguous expression, we can use parentheses. As we've seen above, all operators are binary (except `-` which can also be unary) and we can just place a pair of parentheses around

every subpart of an expression. However, python also has a set of precedence rules, which ensure that the result of evaluating an expression is uniquely defined even when it is not fully parenthesised. The precedence rules are summarised in Chapter 2 of Downey's book (section "Order of Operations"), section 1.7.4 in Punch & Enbody's book, and in the online python documentation.

Write a function `print_expression(exp)` that takes an expression, in the form of a nested tuple as above, and prints out the expression using parentheses only where it is necessary. For example,

- `( (2, '**', 4), '-', 1)` should be printed as `2 ** 4 - 1`
- `( 2, '**', (4, '-', 1) )` must be printed as `2 ** (4 - 1)`
- `( 5, '/', (5, '*', 2) )` must be printed as `5 / (5 * 2)`
- `( ( 5, '/', 5), '*', 2 )` should be printed as `5 / 5 * 2`

Note that to decide if a subexpression needs to be parenthesised, you must compare the precedence of its operator with that of the expression that it is part of.

To keep the function simpler, you can make the same assumptions as in (a) and (b) above (constants are numeric or string values, operators are the standard arithmetic ones).

Exercise 2: Debugging, conditionals and iteration

(a) The following are attempts to define a function that takes three (numeric) arguments and checks if any one of them is equal to the sum of the other two. For example, `any_one_is_sum(1, 3, 2)` should return `True` (because `3 == 1 + 2`), while `any_one_is_sum(0, 1, 2)` should return `False`.

All of the functions below are incorrect. For each of them, find arguments that cause it to *return* the wrong answer.

Function 1

```
def any_one_is_sum(a,b,c):
    sum_c=a+b
    sum_b=a+c
    sum_a=b+c
    if sum_c == a+b:
        if sum_b == c+a:
            if sum_a == b+c:
                return True
    else:
        return False
```

Function 2

```
def any_one_is_sum(a,b,c):
    if b + c == a:
        print(True)
    if c + b == a:
        print(True)
    else:
        print(False)
    return False
```

Function 3

```
def any_one_is_sum(a, b, c):
    if a+b==c and a+c==b:
        return True
   else:
        return False
```

(b) The following are attempts to define a function that computes a sum. Each one uses a `while` loop that may never terminate. The loop may terminate for some arguments, but not for others. For each of the functions, find arguments that cause the loop to never terminate. The arguments should all be numbers, `lower` should be less than `upper`, and `nterms` should be a positive integer (not zero).

Hint: Add `print` calls *inside* the loop to see what is happening. Print the variables that appear in the loop condition, so you can see if they are changing or not (if they are not, then the loop is stuck).

Function 1

```
def integrate(lower, upper, nterms):
    delta = (upper - lower) / nterms
    total = 0
    while lower+delta >= upper:
        area = ((2 - lower ** 2) + (2 - (lower + delta) ** 2)) * delta / 2
        total = total + area
        lower = lower + delta
    return total
```

Function 2

```
def integrate(lower, upper, nterms):
    delta = (upper - lower) / nterms
    total = 0
    while lower < upper:
        area = ((2 - lower ** 2) + (2 - (lower + delta) ** 2)) * delta / 2
        total = total + area
```

```
        delta = (upper - lower) / nterms
        lower = lower + delta
    return total
```

Exercise 3: Sequence indexing

A *sequence type* in python is a data type that has a length and can be indexed. To find the length of a value of a sequence type, we use the `len` function (for example, `len("abcde")`), and to index we use the indexing operator, which is written with the index in a pair of square brackets after the sequence (for example, `"abcde"[1]`). The index must be an integer, and the index of the first element in a sequence is zero. There are three built-in sequence types in python: strings (type `str`), lists (type `list`) and tuples (type `tuple`).

(a) Python's built-in sequence types (and most sequence types defined in other modules) allow indexing from the end, using negative numbers. To make sure that you understand how indexing with positive and negative indices works, write a function `reverse_index(index, length)` that takes an index and a length (both integer values, the length non-negative) and returns the corresponding index in the reverse direction. That is, if `index` is positive (indexing from the beginning of a sequence), the function should return the negative index that marks the same position in the sequence, and if `index` is negative (indexing from the end of the sequence) it should return the corresponding positive index.

To test your function, observe that

```
s[i] == s[reverse_index(i, len(s))]
```

should be `True` for any sequence `s`, as should

```
reverse_index(reverse_index(i, l), l) == i
```

for all valid `i` and `l`.

(b) What happens if the index is not valid (outside the range of indices of a sequence of the given length)? Is it possible to reverse an invalid index?

Exercise 4: Debugging functions on lists

(a) Here is an attempted solution to the programming problem "Counting duplicates" from Lab 4. It has several errors. Can you find what is wrong with it, and fix the errors?

What is the idea behind this function? Does the idea work? (after the errors have been fixed). Does it work for any input sequence?

```
def count_duplicates(sequence):
    count = 0
    sorted_sequence = sequence.sort()
```

```
    index = 0
    while index < len(sorted_sequence) - 1:
        if sorted_sequence[index] == sorted_sequence[index + 1]:
            count = count + 1
            index = index + 1
        return count
```

The number of duplicates is equal to the number of elements in the sequence minus the number of unique elements. Here are some examples:

- `"abracadabra"` has 6 duplicates (4 `a`, 1 `b` and 1 `r`)
- `"mississippi"` has 7 duplicates (3 `s`, 3 `i` and 1 `p`)

(b) Here is another attempted solution to the programming problem "Counting duplicates" from Lab 4. It also has some errors: find and fix! What is the idea behind this implementation? Add comments to describe it. Can it be made to work?

```
def count_duplicates(sequence):
    index = 0
    length = len(sequence)
    while index < len(sequence):
        if sequence[index] in sequence[index + 1:]:
            sequence.pop(index)
        index = index + 1
    return length - len(sequence)
```

(c) The last problem of Exercise 1 from Lab 5 asks you to:

- Create a list of 10 lists of increasing length. The first list should be empty, i.e., `len(mylist[0]) == 0`; the second list should have one element, so `len(mylist[1]) == 1`; and so on.

Here is an attempted solution to this problem. Does it work? If not, why not? How can you fix it?

```
mylist = []
mylist2 = []
for count in range(10):
    mylist.extend(mylist2)
    mylist2.append(count)
print(mylist)
```

Programming problems

Note: If you do not have time to finish all programming problems in the lab, you should continue working on them later (at home, in the CSIT labs after teaching hours, or on one of the computers available in the university libraries or other teaching spaces). On the other hand, if you have extra time during the lab, you can go back to any programming problems in the previous labs that you have not yet finished.

Subsequence find

The `str` type has several methods that are not available for python's other built-in sequence types, for example `str.count`, `str.find` and `str.rfind`. A common feature of these is that they allow searching for (or counting) a substring of a string, not just a single character.

Write functions `find(seq, subseq)` and `rfind(seq, subseq)` that behave like the string methods of the same name, that is, that return the first and last position, respectively, where the second argument occurs, as a consecutive subsequence, in the first argument sequence.

Cumulative sum

(Exercise 10-2 in Downey's book.)

(a) Write a function `cumulative_sum(seq)` that takes a sequence of numbers and returns a list of their cumulative sums. This means that the i:th element in the returned list should be the sum of elements 0 .. i in the argument sequence. Can you implement this function with a single loop over the argument sequence?

Try your function on arguments that are lists, and arguments that are NumPy arrays. What is the result if you apply the function to a 2-dimensional array, like

```
np.array([[1,2,3], [4,5,6], [7,8,9]])
```

What happens if you apply it to a nested list, like

```
[[1,2,3], [4,5,6], [7,8,9]]
```

instead? Why? Are the return values correct in both cases?

(b) You probably implemented the `cumulative_sum` function using a loop. Can you do it without a loop, using recursion instead?

(c) Does the NumPy module already provide a function that does the same as the one you wrote? Have a look at the documentation.