

CSC1010H Tutorial 6 – Recursion

Introduction

Learning outcomes

Skills

- Reason about, devise and implement recursively defined solutions to programming problems.

Knowledge

- Concept of a recursive function, 'making progress' and 'base case'.
- How recursion works: run-time stack.

Question 1 [10 marks]

Do you remember that remarkable species of Mongoose from tutorial 4?

"A remarkable species of mongoose has been discovered, remarkable because it is extremely long-lived and because it has strangely regimented breeding behaviour:

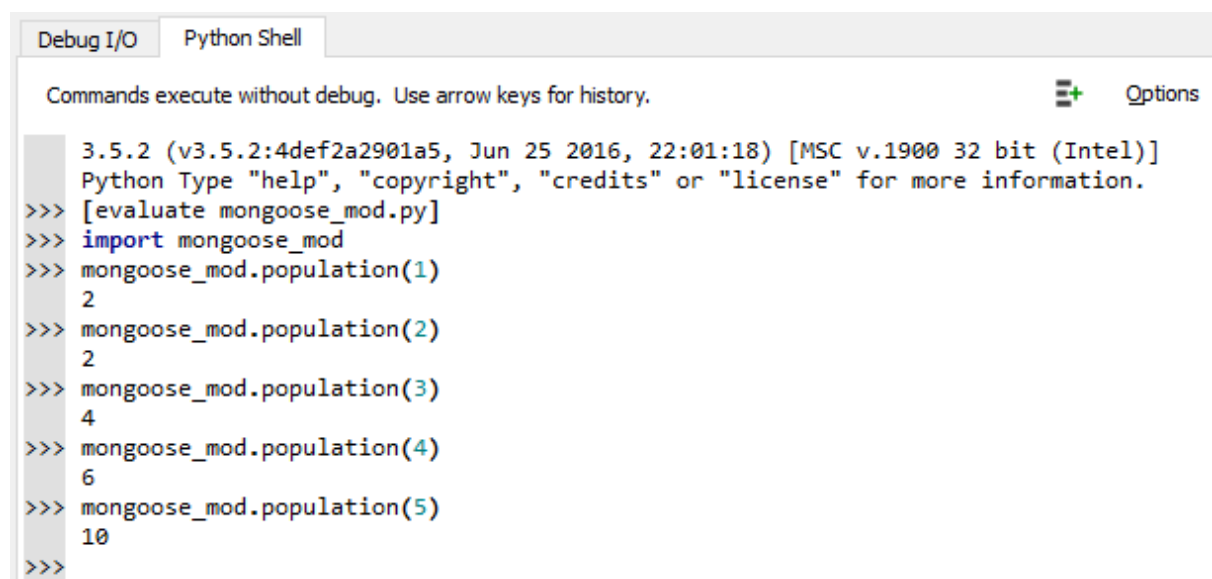
- A new born mongoose takes three months to reach sexual maturity.
- On reaching sexual maturity, a mongoose immediately finds a mate.
- A mongoose mates for life.
- A breeding pair of mongooses (mongeese? mongoosia?) gives birth to a new pair of mongooses (exactly one male and one female) every month after reaching sexual maturity."

The general rule is that the population size, $P(n)$ is:

$$P(n) = P(n - 1) + P(n - 2)$$

Write a recursive Python function called 'population' that has a parameter 'n', and that calculates $P(n)$. Call the module that contains your function 'mongoose_mod.py'.

If everything works correctly then you should be able to replicate the following interaction with the Wing IDE Python Shell:



```
Debug I/O Python Shell
Commands execute without debug. Use arrow keys for history. Options
3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:01:18) [MSC v.1900 32 bit (Intel)]
Python Type "help", "copyright", "credits" or "license" for more information.
>>> [evaluate mongoose_mod.py]
>>> import mongoose_mod
>>> mongoose_mod.population(1)
2
>>> mongoose_mod.population(2)
2
>>> mongoose_mod.population(3)
4
>>> mongoose_mod.population(4)
6
>>> mongoose_mod.population(5)
10
>>>
```

CONTINUED

Question 2 [20 marks]

A palindrome is a string that reads the same backwards and forwards, generally ignoring spaces, punctuation and capital letters; that is, to determine if a string is a palindrome you look only at the alphabetic characters.

For example,

“Able was I, ere I saw Elba.”

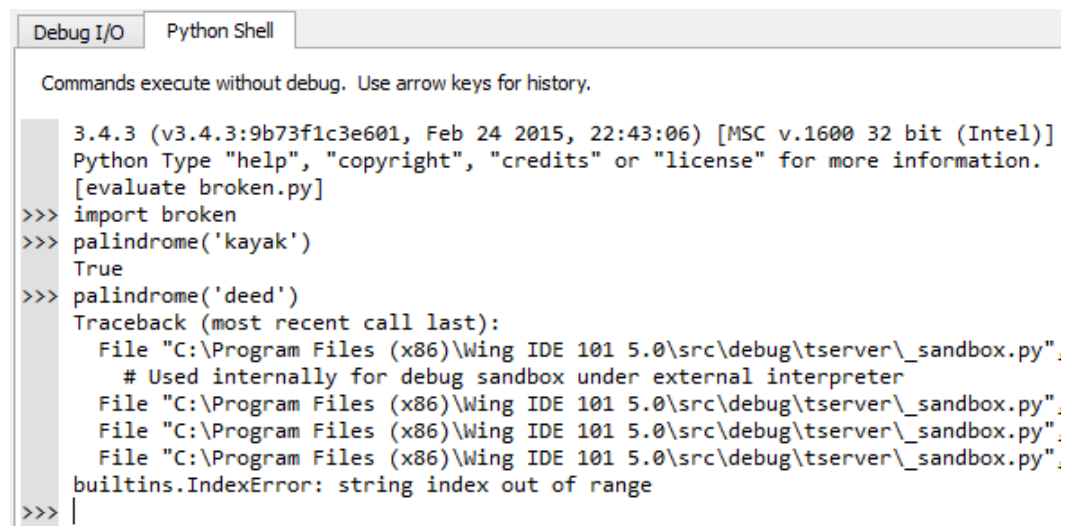
The following recursive Python function is meant to check whether a given sentence or word is a palindrome. It partially works.

```
import string

def palindrome (s):
    if len(s) == 1:
        return s.isalpha()
    elif s[0] in string.punctuation or s[0] in string.whitespace:
        return palindrome(s[:-1])
    elif s[-1] in string.punctuation or s[-1] in string.whitespace:
        return palindrome(s[:-1])
    elif s[0].lower() == s[-1].lower():
        return palindrome (s[1:-1])
    else:
        return False
```

The function is available from the Vula tutorial page. It is in a module called ‘*broken_mod.py*’.

Let us say that we load ‘*broken_mod.py*’ into Wing IDE and then, at the shell enter ‘import broken_mod’ and then ‘palindrome(“kayak”)’ , we get the result ‘True’, which is correct. If, however, we enter ‘palindrome(“deed”)’ , It fails.



```
Debug I/O Python Shell
Commands execute without debug. Use arrow keys for history.

3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)]
Python Type "help", "copyright", "credits" or "license" for more information.
[evaluate broken.py]
>>> import broken
>>> palindrome('kayak')
True
>>> palindrome('deed')
Traceback (most recent call last):
  File "C:\Program Files (x86)\Wing IDE 101 5.0\src\debug\tserver\_sandbox.py",
    # Used internally for debug sandbox under external interpreter
  File "C:\Program Files (x86)\Wing IDE 101 5.0\src\debug\tserver\_sandbox.py",
  File "C:\Program Files (x86)\Wing IDE 101 5.0\src\debug\tserver\_sandbox.py",
  File "C:\Program Files (x86)\Wing IDE 101 5.0\src\debug\tserver\_sandbox.py",
builtins.IndexError: string index out of range
>>> |
```

To figure out why one works and the other doesn’t, we could add in print statements to see exactly what happens. Doing a paper-based simulation, however, is great for developing code comprehension skills.

There are no hard and fast rules on ‘playing computer’. Using ‘palindrome(“kayak”)’ as an example, here is one possibility:

```

palindrome("kayak") : return palindrome (s[1:-1])
>palindrome("aya") : return palindrome(s[1:-1])
>>palindrome("y") : return s.isalpha()
True

```

- Starting with `'palindrome("kayak")'`, we study the code and determine that the result is `'return palindrome(s[1:-1])'`.
- We write this down: the expression `'palindrome("kayak")'`, and the result `'return palindrome(s[1:-1])'`, separated by a colon.
- We figure out that `s[1:-1]` is `"aya"`, so now we consider `'palindrome("aya")'`. This is a recursive call. To denote this we begin our next line with a `'>'`.
- We study the code again, and figure out the result of `'palindrome("aya")'` is `'return palindrome(s[1:-1])'` (again). We write this down.
- We figure out that `s[1:-1]` is `"y"`, so now we need consider `'palindrome("y")'`. This is another recursive call. To denote this we begin our next line with one more `'>'` than last time.
- We study the code again, and figure out that the result is `'return s.alpha()'`. We write this down.
- Now, we know that `'s.isalpha()'` must be true because we know that `s` is `"y"`. So, the overall result of `'palindrome("kayak")'` is `True`.

Your tasks:

1. Construct 'paper-based' traces like the one above for the following expressions:
 - a. `palindrome('deed')`
 - b. `palindrome('at noon, ta')`

These are going to fail.
2. On the basis of your studies, figure out the problem(s) and solutions.
3. Make a correct (fixed) version of `'palindrome'` in a Python module called `'fixed_mod.py'`.

Question 3 [20 marks]

Write a recursive function called `countpairs` that has a parameter, `'s'`, a string. The function will count the number of pairs of repeated characters in a string.

Pairs of characters cannot overlap e.g. `'aaa'` counts as one pair of `'a's` followed by a single `'a'`.

Call the module that contains your function `'pairs_mod.py'`.

Use the Wing IDE Python shell to check your function e.g.

```

Debug I/O Python Shell
Commands execute without debug. Use arrow keys for history.
3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 18:41:36) [MSC v.1900 64 bit (AMD64)]
Python Type "help", "copyright", "credits" or "license" for more information.
>>> import pairs
>>> pairs.countpairs('hello, Salaama')
2
>>>

```

HINT: The base case is `'If len(s)<2 then return 0'`.

The return of Frogs and Toads

Do you remember 'Frogs and Toads', the simple puzzle game from assignment 4?



(Icon courtesy of icons8.com.)

There is an empty square between the frogs and the toads. The challenge is to move the creatures, one at a time such that the positions are reversed (all the frogs on the right, and all the toads on the left) e.g.



Here is a reminder of the:

- There is only ever one empty square.
- A square is either the empty square, or it contains exactly one frog or toad.
- A frog or toad facing the empty square can hop into it.
- A frog or toad can hop over another frog or toad if the empty square is on the other side.
- A frog can only hop from left to right.
- A toad can only hop from right to left.

Your challenge is to develop a recursive 'Frogs and Toads' puzzle solver called 'solver.py'.

Here's a sample of expected program behaviour:

```
Enter the number of frogs:
```

```
1
```

```
Enter the number of toads:
```

```
2
```

```
Here is the initial state:
```

```
|Frog|      |Toad|Toad|
```

```
Here is the solution (in reverse):
```

```
|Toad|Toad|      |Frog|
```

```
|Toad|Toad|Frog|      |
```

```
|Toad|      |Frog|Toad|
```

```
|Toad|Frog|      |Toad|
```

```
|      |Frog|Toad|Toad|
```

```
|Frog|      |Toad|Toad|
```

The solution is printed in reverse order, and consists of a series of states i.e.

Last state

Second to last state

.....

Second state

First state

We have arranged it this way to make the problem easier to solve.

You may remember that, in assignment 4, we printed a puzzle state using this format:

```
  1    2    3    4
Frog   ToadToad
```

The indices helped the player select which frog or toad to move. We don't need that here as the computer is the player, however, we've put in vertical spacing bars to make the puzzle squares easier to see.

We have broken the problem of writing *solver.py* into two parts. For the first part, you must write a Python module containing helper functions. For the second part you will write the recursive code that actually solves frog and toad puzzles.

We will assume that a puzzle state is represented using a list of strings.

- If a square contains a frog then the corresponding location in the list contains the string 'Frog'.
- If a square contains a toad then the corresponding location in the list contains the string 'Toad'.
- An empty string, '', is stored in the location corresponding to the empty square.

For example:

```
['Frog', 'Frog', '', 'Toad', 'Toad']
```

Question 4 [15 marks]

Write a Python module called 'solver_mod.py' that contains the following functions:

- `make_state(num_frogs, num_toads)`

Create an initial puzzle state consisting of a list containing the string 'Frog' a `num_frogs` number of times and then an empty string, "", and then the string 'Toad' a `num_toads` number of times.

For example, the expression `make_state(3, 2)` will return `['Frog', 'Frog', 'Frog', '', 'Toad', 'Toad']`.

- `find_space(state)`
Obtain the index of the empty puzzle square (i.e. the index of the empty string in the list.)
For example, assuming `st=['Frog', 'Frog', 'Frog', '', 'Toad']`, the expression `find_space(st)` will return 3.
- `is_frog(state, index)`
Return `True` if there is a frog at the given index, otherwise return `False`.
- `is_toad(state, index)`
Return `True` if there is a toad at the given index, otherwise return `False`.
- `move(state, index)`
Return the new state, S' , produced by moving the frog or toad at the given index.

For example, assuming `st=['Frog', 'Frog', 'Frog', '', 'Toad']`, the expression `move(st,1)` will return the new list `['Frog', '', 'Frog', 'Frog', 'Toad']`.

NOTE: Your function should begin by creating the new list using statement: `'newState = list(state)'`.

- `print_state(state)`
Print the given state as a sequence of strings separated by vertical bars, '|'.
For example, assuming `st=['Frog', 'Frog', 'Frog', '', 'Toad']`, the expression `print_state(st)` will output: `'|Frog|Frog|Frog| |Toad|'`.
- `is_win(state)`
Return `True` if state consists of a series of toads, followed by a space, followed by a series of frogs, otherwise return `False`.

Question Five [35 marks]

The recursive algorithm for this problem is a version of what is called a 'depth-first search', and is based on the idea of, given a puzzle state, *S*, determining if it is solvable.

We want to write a function called '*solvable*'. Given a puzzle state, *S*, if *solvable(S)* is *True*, then *S* is solvable i.e. a series of moves can be made to get to the winning state (a series of toads, followed by a space, followed by a series of frogs). If *solvable(S)* is *False*, then it is not solvable i.e. a series of moves cannot be made to get to the winning state.

The *solvable* function is recursive. Here is how it functions:

- If *S* represents a winning state (a series of toads, followed by a space, followed by a series of frogs, i.e. if *is_win(S)* then, yes, it is solvable, return *True*.
If *S* does not represent a winning state, then the question is whether any moves can be made.
- If moves can be made, taking each in turn, make the move to get a new state, *S'*, and then - this is the recursive step - determine if *solvable(S')*. If *S'* is solvable then *S* is solvable, return *True*.
- If no moves can be made, e.g. say *S* is `'| |Frog|Frog|Frog|Toad|'`, then *S* is not solvable, return *False*.
- The final rule (the one that gets the desired output) is that if *S* is solvable then print it before returning *True*.

Given a state, *S*, how do we determine if any moves can be made? The easiest way is to find the empty square/space, and look at the frogs and toads on either side. If there is a frog one square or two squares to the left, then it can be moved. If there is a toad one square or two squares to the right, then it can be moved.

Given a state, *S*, and assuming *find_space(S)* is *N*:

#	Precondition	Action
1	<i>is_frog(S, N-1)</i>	<i>S'=move(S, N-1)</i>
2	<i>is_frog(S, N-2)</i>	<i>S'=move(S, N-2)</i>
3	<i>is_toad(S, N+1)</i>	<i>S'=move(S, N+1)</i>
4	<i>is_toad(S, N+2)</i>	<i>S'=move(S, N+2)</i>

Complete the *solver.py* program by writing a *solvable* function and a suitable *main* function.

Marking and Submission

Submit the *mongoose_mod.py*, *pairs_mod.py*, tracing text file, *fixed_mod.py*, *solver_mod.py*, and *solver.py* contained within a single .ZIP folder to the automatic marker. The zipped folder should have the following naming convention:

yourstudentnumber.zip

Marking Guide

1. Mongooses implementation	10
2. Palindrome tracing (12 marks manually awarded)	20
3. Pairs	20
4. Frogs and Toads utility module.	15
5. Frogs and Toads solver.	35
Total	100

END