Name: Date:

Lesson 8: Algorithms

Algorithms

In Maths and Computing, an **algorithm** is a step by step list of instructions for how to perform a calculation. An algorithm is like a cooking recipe – anyone following the instructions should be able to produce the same result. Algorithms are normally implemented using functions. Here's an example of an algorithm to find the largest number in a list of integers:

```
def findLargestNumber(ls):
    big=0
    for i in ls:
        if i > big:
            big=i
    return big

numbers=[1,23,43,21,5,6,98,43,2,9]
print(findLargestNumber(numbers))

--- OUTPUT ---
98
```

In this algorithm, the steps are as follows:

1. Set `big` to be 0
2. Iterate the list of integers `ls` with a `for` loop
3. For each integer `i` in list `ls` if `i` is bigger than current `big`, set `big` to that number
4. Carry on until all numbers in `ls` have been looked at
5. Return the current value of `big` which will be the biggest number found in `ls`

When you're trying to develop an algorithm in code, it's a good idea to try and write or at least sketch out the steps involved so that you are clear about the structure of the code you're going to write.

TRY IT OUT #1: Try creating an algorithm to find the longest word in a list of words.

Here's a fully worked solution:

```
def findLargestWord(l):
    big=''
    for w in l:
        if len(w) > len(big):
            big=w
    return big

l=['this','is','a','list','of','monkeys']
print(findLargestWord(l))

--- OUTPUT ---
monkeys
```

Another good example of an algorithm was last week's implementation of `topLettersByCount(s)` to find the most commonly occurring letters in a long text string.

TRY IT OUT #2: Review the solution provided to last week's Code Quest. Write out the algorithm steps after reading and understanding the code.

Tuples

A tuple is a fixed ordered list of elements. You can consider it to be a fixed or **immutable** list. If you try to modify a tuple, you will end up creating a new one as with other immutable types such as string. A tuple is created by using the normal bracket operators:

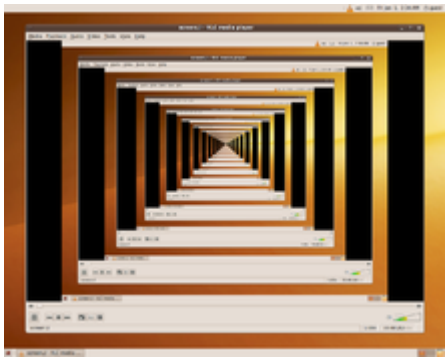
```
t=(1,2)
type(t)

--- OUTPUT ---
<type 'tuple'>
```

TRY IT OUT #3: Try creating a tuple and then a list of tuples in the Python interpreter

Recursion

Recursion is the process of repeating items in a **self-similar** way. In programming it means writing a function that calls itself!! That might sound crazy – it's like looking in two mirrors and seeing your reflection go onto infinity. Indeed there's a famous joke about recursion:

"To understand recursion, you must understand recursion."



You may wonder, how does it ever return from calling itself? The answer lies in setting an appropriate termination condition. Every time you call the same function, its work is added to a memory stack. When that termination condition is reached, the stack unwinds as all the instances of the function are returned from termination condition back out. Consider an algorithm that reverses a string. It might be broken down as follows for string `s`:

1. Take the string s and break it into two parts s[0:1] which is the first character and s[1:] the remainder
2. Return reverse of remainder appended with s[0:1]. So eg. "monkey" => reverse("onkey")+"m"
3. Keep going as long as the length of string to reverse > 1
4. Terminate if the length of s is 1 (because that is the smallest string you can reverse)
5. Unwind the recursion stack by returning from each recursive function call

Here is the code that implements this algorithm using recursion. Study it carefully!

```
def rreverse(s):
    if len(s) == 1:
        return s
    else:
        return rreverse(s[1:]) + s[0:1]

print(rreverse("This is a string"))

--- OUTPUT ---
gnirts a si sihT
```

TRY IT OUT #4: Try creating a recursive function for calculating factorial.

A fully worked solution for factorial done using recursion in Python is below. Note the termination condition is n less than or equal to 1 which results in return of 1. Otherwise we return `n x factorial(n-1)`.

```
def rfactorial(n):
    if n <= 1:
        return 1
    else:
        return n * rfactorial(n-1)

print(rfactorial(1))
print(rfactorial(2))
print(rfactorial(3))
print(rfactorial(4))
print(rfactorial(5))

--- OUTPUT ---
1
2
6
24
120
```

Bringing it all together ….

Congratulations! You've made it to the end and survived recursion! We've covered a huge amount of material in this short course! It's only a starting point and there's plenty more to learn as you progress. The most important thing to do at this point is to spend some time now reviewing all the provided material and making sure you understand:

- **Basic data types**: string, int, list, dictionary, bool, float, tuple
- **Basic control structures**: for loop, list comprehensions, if/elif/else
- **Algorithms**: how to break down problems into a list of steps
- **Functions**: useful little blocks that we can build bigger blocks on.

The best way to get better is by actually trying to write code. There are a lot of resources on the web to support you. A particularly good option if you're comfortable with everything covered so far is to enrol on the Codecademy Python course:

http://www.codecademy.com/en/tracks/python

SECRET CODE QUEST!

Now you choose! Let's brainstorm some ideas. That's what happens in a "Python Dojo".