# Semester 2, 2018: Lab 3

S2 2018

## Lab 3

*Note:* Like the last lab, there may be more material in this lab than you can finish during the lab time. If you do not have time to finish all exercises (in particular, the programming problems at the end), you can continue working on them later. You can do this at home (if you have a computer with python set up), in the CSIT lab rooms outside teaching hours, or on one of the computers available in the university libraries or other teaching spaces.

### Exercise 0: Truth values

A truth value, or boolean value is a value that has only two possible states: `True` and `False`. You get a boolean value as a result of using a comparison operator. Both `True` and `False` are reserved words in python, and they are different to strings such as "True" and "False".

Start up a shell, and try evaluating following expressions. Make sure you understand the results you obtain:

```
In [1]: 5 > 4
Out [1]: ...

In [2]: type(5 > 4)
Out [2]: ...

In [3]: "aab" > "ab"
Out [3]: ...

In [4]: type("aab" > "ab")
Out [4]: ...

In [5]: type("False")
Out [5]: ...
```

You can combine boolean values with *boolean operators* `and`, `or` and `not`.

```
In [6]: 5 > 4 and 4 > 9
Out [6]: ...

In [7]: not 4 > 9
Out [7]: ...

In [8]: True or False
Out [8]: ...

In [9]: True or True
Out [9]: ...

In [10]: type(True or False)
Out [10]: ...
```

Note: python allows you to write some compound boolean expressions without a boolean operator:

```
10 < x <= 20
```

means the same as

```
10 < x and x <= 20
```

The comparisons can be both strict, both non-strict, or a mixture of strict and non-strict.

## Exercise 1: Programs with conditional branching

**From here on, you should switch to writing your code in a file rather than typing it into the shell.**

You can use boolean values to make decisions in `if` statements - this is called *conditional branching*.

### Exercise 1(a)

The lecture slides contain the following function for determining the grade associated with a final mark:

```
def print_grade(mark):
    if mark >= 80:
        print("High Distinction")
    if mark >= 70:
        print("Distinction")
    if mark >= 60:
        print("Credit")
```

```
    if mark >= 50:
        print("Pass")
    else:
        print("Fail")
```

Copy the function to a file so that you can run it, and test it with a range of different values (keeping in mind that the mark should be a number between 0 and 100) so that you see all the possible outputs. As we saw in the lecture, this function does not perform as intended, and we saw one way to fix it, by replacing each test with a compound test expressions (for example `mark < 80 and mark >= 70`).

Rewrite the function so that it works correctly using only simple comparison expressions (no boolean operators), if and else statements.

**Exercise 1(b)**

There is one more variant of the `if` statement in python that we only briefly mentioned in the lecture: using `elif`. `elif` is an abbreviation of *else if*, and it allows you to write `if` statements in the `else` suite in a more compact way.

```
if test_expression_1:
    ...suite 1...
elif test_expression_2:
    ...suite 2...
else:
    ...suite 3...
```

is equivalent to writing

```
if test_expression_1:
    ...suite 1...
else:
    if test_expression_2:
        ...suite 2...
    else:
        ...suite 3...
```

However, you can write any number of `elif` parts into an `if` statements, not just one.

Rewrite the function `print_grade` using `elif`, so that it uses only simple comparison expressions (no boolean operators) and still does the right thing.

### Exercise 1(c)

The median of three numbers, a, b and c, is the one that ends up in the middle when the numbers are sorted in increasing order. For example, the median of -2, 7 and 9 is 7, and the median of 7, 9 and -2 is also 7.

Write a function `median` with three parameters `a` `b` and `c`, which returns the median of its three arguments. (Of course, the function only works if all three arguments are numbers... or does it? What happens if you call it with three strings? What happens if you call it with a mix of numbers and strings?)

Note that there are many different ways to solve this problem.

**Testing your function**. You can test the function with sets of small numbers (like 1,2,3); if it works for small numbers, it should work just as well for large numbers. Note that the median of three numbers is the same no matter in which order they are given to the function; that is, `median(1,2,3)`, `median(1,3,2)`, `median(2,1,3)`, `median(2,3,1)`, `median(3,1,2)` and `median(3,2,1)` should all return the same result. What happens if you call your function with two or three copies of the same value? (such as `median(2,1,2)` or `median(1,1,1)`).

## Exercise 2: The robot simulator revisited

First, if you have not done so already, download a copy of the robot.py module and save it in your working directory for this lab. Make sure that you can import the module: the python interpreter will look for it in the current working directory.

Remember that you must start by initialising the simulation:

```
robot.init()
```

(If you need more of a reminder about how to run and program the robot, revisit Lab 1.) For these problems, you will need to use the robot's box sensor. The command `robot.sense_color()` returns the colour of the box in front of the sensor, as a string (a value of type `str`). It will return an empty string (`''`) if there is no box in front of the sensor. For more about how the box sensor works, see last week's lecture on branching.

### Programming problems

1. In an early lecture, we introduced the problem of stacking boxes lined up on the shelf into a tower. Even though we could define functions that encapsulate the main steps of picking up, stacking and so, we still needed to write different programs, with a different number of repetitions, for different numbers of boxes. (For example, here are programs for stacking 3 boxes and stacking 5 boxes.)

   Write a single function, `make_tower`, that stacks any number of boxes that stand in a line on the shelf into a single tower.

   To create problems with different sizes, you pass extra arguments to the `init` function, for example:

```
robot.init(width = 7, boxes = "flat")
```

The "width" is the number of spaces on the table; because the initial setup leaves one empty space between each pair of boxes (otherwise the robot would not be able to pick them up), you need a width of 2*n - 1 to make a problem with n boxes (i.e., width 5 gives you 3 boxes; width 7 gives you 4 boxes; and so on).

There are two important questions to consider in the designing an algorithm for this problem:

What is the sequence of commands that is repeated for every box added to the stack?

How do you determine when to stop? The robot's sensor can only detect the presence of boxes; how can you tell if it has reached the end of the table?

2. In last week's lecture, we developed both a recursive and an iterative function for counting the number of boxes in a stack.

   Following the same idea, write a function that finds the position of a box with a given colour in a stack. That is, your function should look like this:

   ```
   def find(colour):
       ...
   ```

   and should return the position (height above the shelf) where a box of that colour is. If there is no box of the specfied colour, the function should return a distinct value, such as -1. (We choose -1 because it can never be the position in a stack, so we can distinguish it from the case where a box has been found.)

   Note that there are two base cases: either the box is found, or the robot has searched all the way through the stack.

   To test your function, you need to set up simulations with a tall stack of different colours. You can do this by passing additional arguments to the `init` function, for example:

   ```
   init(height = 5, boxes = [["black", "blue", "white", "red"]])
   ```

   The `height` argument specifies the maximum height that the lift can go.

## Exercise 3: The square root algorithm

The Babylonian algorithm is a method of computing square roots of numbers. The method was recorded as far back as the 1st century, though the name suggests it is older than that. The algorithm can be derived as a special case of the Newton-Raphson method for solving an equation of the form f(x) = 0.

The Babylonian algorithm works as follows: We want to compute the square root of some number a. First, we take a guess: pick some number x. Then check how good is our guess, by computing the (absolute) difference abs(x2 - a). If it's close enough (for example, less than 10-6) we're done. Otherwise, calculate an improved guess (x + a/x) / 2; this replaces the previous value of x and we repeat from the test.

Implement a function that computes square roots using the Babylonian algorithm. From the algorithm's description, you may already have guessed that using a `while` loop is a good way to go.

**Testing your function**. The `math` module provides a square root function, `math.sqrt`, so you can test your function by comparing its result with what `math.sqrt` returns. Try changing the threshold for when the answer is close enough and see what effect this has on the values returned by your function.

To get a better understanding of what is happening, you can also try adding a `print` call inside the loop in your function, so that you can see how the guesses are updated in each iteration. They should be converging towards the correct answer.

(You can find this example, and it's solution, in both text books. It's in Chapter 7 in Downey's book, and Chapter 3 in Punch & Enbody's book. Note that Punch & Enbody use `input` for console input to get test values. This is totally unnecessary - write a function instead.)

## Exercise 4: Sums and divisors

A *divisor* of a (positive integer) number n is a number that divides n evenly. For example, 1, 2, 3, 4, 6 and 12 are all divisors of 12. (They are also all the divisors of 12, meaning there are no others.) A divisor that is not 1 or the number itself is usually called a "proper" divisor. As you probably know, a prime number is a number that has no divisors other than 1 and the number itself. A *prime factor* of n is a proper divisor of n that is also a prime number.

Any positive integer can be constructed as the product of its prime factors, though some factors may need to be repeated some number of times. For example, 12 = 2 * 2 * 3. The number of repetitions of each prime factor in the product is known as its *multiplicity*.

1. Write a function `print_factors` that prints out the prime factors of a number; the number is the argument to the function. (We choose to print, rather than return, the factors here because we don't know how many there will be for any given number and we have not yet seen how to return data structures with a varying number of parts from a function.)

2. As part of writing the `print_factors` function, did you write a separate function to identify prime numbers?

3. Modify your function so that it finds (and prints) the multiplicity of each factor as well.

4. A number is called *perfect* if it is equal to the sum of its divisors, including 1 but excluding (of course!) the number itself. (Note that this is the sum of all divisors, not just the prime ones. Each divisor is added only once, regardless of multiplicity.) Write a function to check if a given number is perfect.

5. Write a loop that prints out all perfect numbers between 1 and some upper limit (for example, 100).

For these questions you may wish to refresh your memory about the `//` and `%` operators.