Lab 02

Branching and Looping

- `if`, `elif`, and `else`
- `while` and another `else`
- Iterating with a `for` loop
- Counting loop with `range`
- Relational and logical operators
- `tuples`

else.py

```
 1 #!/usr/bin/env python
 2 """Demonstrates if/elif/else and while/else."""
 3
 4 number = 25
 5
 6 if number < 10:
 7     print number, 'is small'
 8 elif number >= 1000:
 9     print number, 'is big'
10 else:
11     print number, 'is medium'
12
13 if 10 < number < 50:    # "and" is assumed
14     print "number is in"
15 # Alternate syntax since 2.5 -- all one line but less readable.
16 print number, "is",
17 print "small" if number < 10 \
18               else "big" if number >= 1000 \
19               else "medium"
20 # else can also occur in a loop
21 div = 2
22 while div * div <= number:
23     if number % div == 0:
24         print number, 'is divisible by', div
25         break
26     div += 1
27 else:
28     print number, "is prime"
```



```
$ else.py
25 is medium
number is in
25 is medium
25 is divisible by 5
```

range — Built-in Function

```
>>> range(10)

 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> range(5, 10)

 [5, 6, 7, 8, 9]

>>> range(2, 11, 2)

 [2, 4, 6, 8, 10]

>>> range(10, 0, -1)

 [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

 >>>
```



```
range([start=0,] almost_end[, increment=1])
```

All the same:

```
range(10)
range(0, 10)
range(0, 10, 1)
```

counting_loop.py

```
1 #!/usr/bin/env python
2 """ Demonstrates a for loop """
3
4 for num in range(5):
5     print num, "* 2 =", num * 2
```

```
OUTPUT:
$ for_loop.py
0 * 2 = 0
1 * 2 = 2
2 * 2 = 4
3 * 2 = 6
4 * 2 = 8
```

Relational Operators in Python:

< means less than

> means greater than

<= means less than or equal

>= means greater than or equal

== means equal

!= means not equal

Logical Operators:

`and` means and

`or` means or

`not` means not

Lab 02 – Exercises:

1. How would you produce the following using the `range` operator?

   ```
   [3, 6, 9, 12]
   [-10, 100, 210]
   [-1, -3, -5, -7]
   ```

2. Write a script to produce this output using `range` and `for`:

   ```
   10 9 8 7 6 5 4 3 2 1 BLASTOFF!!!
   ```

3. Try this in the interpreter:

   ```
   >>> for ch in "Howdy":
   ...     print ch
   ...      <-- Here, hit the return key to
                finish the indented block

   >>> for num in (2, 4, 16):
   ...     print num
   ...
   ```


   Strings and comma-separated objects, as well as many other Python objects, can be iterated with the `for` and `in`.

   If the comma-separated objects are not wrapped with `[]` or `{}`, but may be wrapped with `()`, they are called *tuples*.

   If the comma-separated objects are wrapped with `[]` or `{}`, they are other collection objects with terrific facilities, and we'll study them soon. And try this:

   ```
   for thing in (2, "hat", (0, 1)):
       print thing
   ```

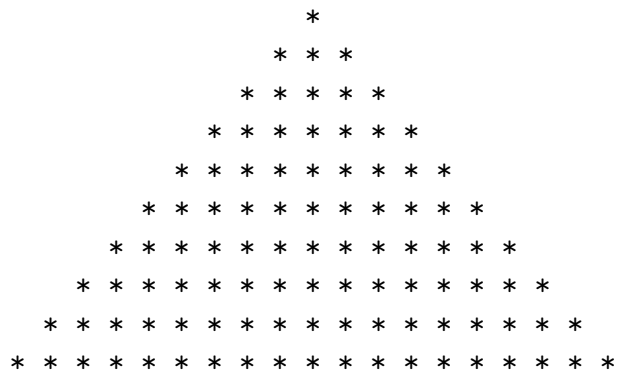
   A tuple can contain any sort of object, even nested tuples.

4. Use a for loop and a tuple of strings to produce:

   ```
   Hi ya Manny!
   Hi ya Moe!
   Hi ya Jack!
   ```

   Do it without duplicating any code or data to maximize robustness.

5. (Optional) Write a script that produces this pattern:



Can you find an easier way? Hint: Have another look at exercise Lab1.2.

6. (Optional) Print the decimal equivalent of a binary string. Test with `"1011"`.

```
Binary string: 1011
Decimal equivalent: 11
```

Try it using a for-loop and a while-loop.

Then, (**not optional**), use the `help` facility at the interpreter prompt to learn about the built-in function `int`:

```
>>> help(int)
```

Only read a few lines until you discover the Pythonic way to do this exercise.