Workshop 3: Loops, Algorithms and Root Finding

10 November 2020

By the end of the session you will:

- know what an algorithm is
- write several algorithms using a Python programs that use conditionals and loops

If you have any questions during this session please ask. You should complete all the exercises before next week.

Remember: *you learn by doing! – you learn Python by writing programs!*

Loops

Loops are methods to repeat a block of code. There are two main types of loops, 'for loops' and 'while loops'.

1. **for loops**
   `for` loops are used when the number of repetitions is known. A `for` loop has the following structure:

   ```
   for <iterator> in <iterable>:
       indented statments
       more indented statements
   pass #  end of loop: note that the indentation stops.
   Program continues
   ```

   The iterator is a dummy variable – typically an index – whose value sequentially takes on the values of the iterable. The iterable is typically a range of numbers (e.g., 1 to 100) but can be something more general.

   There are two rules you ***must*** follow:

   1. You must use a colon (:) at the end of the first for statement.
   2. Loops must be *indented*, normally by four spaces. The end of the loop is marked by the end of the indentation

   Sometimes people write `pass`, with no indentation, as a helpful end-of-loop marker, but this is not needed – it is the end of the indentation that is important. (The 'pass' statement itself does nothing.)

Example:

```
# This is a comment statement.  It is here to document your program
# Comment statements are a little like one-line docstrings, and can go anywhere.
fruit_list = ['apple', 'orange', 'pear', 'cherry']
for fruit in fruit_list:
    print("Fruit is ", fruit)     # This is a print statement.
pass
```

We can also use a function called `enumerate` to get the position when iterating through a list.

```
fruit_list = ['apple', 'orange', 'pear', 'cherry']
for i, fruit in enumerate(fruit_list):
    print(f"The {i} fruit is {fruit}")
```

This also demonstrates the `format` method of strings, allowing you to place variable text inside strings. See the Python documentation at:

https://docs.python.org/3.6/library/string.html#formatstrings

for more details on formatting.

If you want to get more information on a function that you don't understand, type the name of the function followed by a questionmark in the console, such as:

```
print?
enumerate?
range?
```

The `range` method is probably the most common way of forming a for loop, described below.

If you don't understand the `enumerate` statement, skip it and come back to it later.

2. **while loops**
   A `while` statement repeats a block of code until a condition has been met. While loops also need a colon and are indented, and so have the following format:

```
while <condition>:
    <statements>
pass
```

Example:

```
counter = 0
while counter < 5:
    counter = counter + 1
    print('Counter =', counter)
pass
```

It is easy to write a `while` loop that never stops, so be careful!

More about For Loops

The built-in python function `range(stop)` is one of the most useful and most common ways to do a `for` loop. This function will create a range object that starts at 0 and goes to the value stop. For example, `range(4)` will create a range that includes `[0, 1, 2, 3]`, and `range(2,4)` will start at and end at 3. Note that the loop goes from the start value to the end value -1, not the end value itself!

Try out `a = range(10)` and `b = range(1,10)`. How are they different?
Examples:

```
for i in range(2,8):
    print(i)
pass
```

You can also skip numbers, like this:

```
for i in range(4,13,2):
    print(i)
pass
```

Write a program that calculates the change in a person's weight on the moon over a period of 25 years where they gained 1kg mass each year. The formula to compute moon weight (M) from earth weight (E) is:

$$M = E \times .165 \tag{1}$$

The code can be found below – but is it correct? How should it be changed to be correct?:

```
"""Compute weight on moon whilst gaining 1kg/year on earth."""
earth_weight = 52.0
for year in range(0, 25):
    current_weight = earth_weight + 1
    moon_weight = earth_weight * 0.165
    print('In year {} your weight is {}'.format(year, moon_weight))
```

1. Using the Spyder editor, reproduce this program and save it as `moon.py`.

2. Execute the program using the `run module` functionality in Spyder. Do you get the output that you expected?

3. Update the program so only the first 15 years are considered. Save and execute the program. Do you get the output that you expected?

4. Remove the ':' and run the program. Do you understand the error message? Note the line on which the error occurs.

5. Update the program so only the final weight is printed to the screen.

6. Update the program so that it prints out a warning if your moon weight exceeds some maximum value (determined by you).

7. Put comments into the code that explains what is initialized, what the loop does, and why you use if statements (if you use them).

   - Remember what you were told in lectures about blocks of code.
   - Remember how to use conditionals (if statements).

While Loops

The following is a program that will loop until an integer no longer meets a specified condition. The code can be found below:

```
x = 45
while x < 50:
    x = x + 1
print(x)
```

1. Reproduce this using the Spyder editor and save it.

2. Execute the program. Do you get the output that you expected?

3. Update the program so x increments by 2 each time. Save and execute the program. Do you get the output that you expected?

4. Remove the ':' and run the program. Do you understand the error message?

5. Update the program so two variables, x and y, are considered. x and y should start with the values 5 and 20 respectively. On each iteration, x should increase by 1 and y decrease by 2. The while loop should stop when x is greater than y.[1]

[1] **Hint:** If you get stuck in a loop, you can exit using Ctl + C.

Algorithms

An **algorithm** is a process, set of instructions, or set of rules used in a computation. For example, if you buy something that has to be assembled before you can use it, you often get a set of instructions, and that is an algorithm. A recipe is an algorithm.

The program you wrote to compute your weight on the moon and to estimate the square root of a number are algorithms.

1. Some standard types of algorithms (from the Table of Contents of "Numerical Recipes" by William Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery) (a) Solution of Equations; (b) Interpolation and Extrapolation; (c) Integration of Functions; (d) Evaluation of Functions (such as infinite series); (e) Special Functions (such as Bessel Functions, Integrals of sines and cosines); (f) Sorting (searching efficiently); (g) Random Numbers; (h) *Root finding and non-linear equations*; (i) Finding Minimums or Maximums; (j) Finding Eigenvalues and Eigenfunctions; (k) Fourier and Spectral Applications; (l) Statistical descriptions of data (mean, variance, skewness); (m) Modelling of Data (like fitting data to straight lines); (n) ODE Integration Methods; (o) Solution of Two-point boundary value problems; (p) Solution of Partial Differential Equations; (q) ...and many more!

   If we write a code to find the square root of x, we were using a simple algorithm since we were searching for the solution to the equation $y \times y - x = 0$.

2. There can be many different kinds of algorithms to accomplish the same task. Choosing which one to use will have to do with aspects such as efficiency, accuracy, and readability. For example, consider the following method to compute an approximation to the square root of x.

   (a) **Exhaustive Enumeration** or "brute force search" is a general (but often very inefficient!) method of finding the solution to a problem. For the problem of finding the square root of x, we begin with a guess, change it by a small amount, check, and continue until it's close enough. Here is demonstration code from the class reference book (*Introduction to Computation and*

Programming using Python, by John Guttage, Figure 3.3) that finds the square root of x using exhaustive enumeration:

```
"""Find square root of number using exhaustive enumeration."""
x = 25.0
epsilon = 0.001
step = epsilon**2
num_guesses = 0
tot_guesses = 1e7
ans = 0.0
while abs(ans**2 - x) >= epsilon and num_guesses <= tot_guesses:
    ans += step
    num_guesses += 1
print(f'Num Guesses: {num_guesses}')
if abs(ans**2 - x) >= epsilon:
    print(f'Failed to find sqrt of {x}'
else:
    print(f'{ans} is close to sqrt of {x}'
```

Notice the += operation. This will advance the variable. For example, `num_guesses += 1` is the same as `num_guesses = num_guesses + 1` and will add one to num_guesses every time the statement is encountered.

Exercises

These exercises should be completed before the next week. For each program you write you should make sure you test each aspect (e.g., each branch when using conditional statements) and ensure that you have used suitable comments.

1. Edit `moon.py` so that it works properly! Then edit it again so that the individual only gains weight if the year is an even number.
   - The program should print out the weight of the individual each year.

2. Write a program `fibonacci.py` that prints out the first 10 numbers in the Fibonacci sequence.[2,3]

3. Copy and debug the program for finding the square root using the method of Exhaustive Enumeration. Give it a name like square-root_exhuasted.py.
   (a) Add comments using the hash # mark that describes what each part of the code does.
   (b) How many iterations does it take to find the square root of 25?
   (c) What happens if you make x bigger?
   (d) How many iterations does it take if x=12000?
   (e) How many iterations will it take for x=25 if we make the guess more accurate by setting epsilon=.001?

4. In this exercise use a new (and better!) algorithm, one attributed to Heron of Alexandria, to approximate the square root of a number. The algorithm is:
   Step (i) Suppose we want to know the square root of 25.
   Step (ii) Start with a guess, g, which you initialize.
   Step (iii) If $g \times g$ is close enough to x, stop and say that g is the answer. You will be the one to decide how close is close!
   Step (iv) If $g \times g$ is not close enough, then update the guess, g to a new value by averaging g and x/g, i.e. $(g + x/g)/2$. That is, we set:
   ```
   g = (g + x/g)/2
   ```

[2] **Hint:** You may want to use a structure similar to `moon.py`.
[3] **Hint:** If you don't remember what the Fibonacci sequence is, look it up!

Step (v) Using this new guess (again called g) repeat the process until $g \times g$ is close enough to x.

Using Spyder write a program to find the square root of all numbers from 1 to 25. Thus:

(a) Create a new program. Call this `squareroot_heron.py` or similar.

(b) Write an 'outer loop' that goes from 1 to 25. This is best done using a 'for loop' and 'range'.

(c) For each number in the loop, find its square root. To do this, you should use another loop to perform the iteration.[4]

(d) The inner loop can be a for loop or (better) a while loop.

(e) In each iteration, test to see if you have found the square root to a good accuracy, and then exit the loop.

(f) In a while loop, the test can be part of the while statement. If you use a for loop, you will need to provide a separate test within the loop. For example, test that $|g^2 - x| < \epsilon$ where ϵ is a small number.

(g) Use the function `abs(f)` to get the absolute value of `f`.

(h) Print the original number and its square root.

[4] Having a loop inside another loop is called a **nested loop**.