Semester 2, 2019: Lab 2

S2 2018

Lab 2

Last week's lab had an introduction to the CSIT lab computing environment, and some (robot) programming problems. If you have any questions or difficulties with this, make sure you talk with your tutor during the lab.

In the same way as you did in Lab 1, we recommend that you create a new directory (say `comp1730/lab2`) for the programs you will write in this lab.

Remember: When you save your python program to a file, the file name must end in ".py".

Note: This lab is longer than the previous one, and we do not expect everyone to complete it during the lab time. If you do not have time to finish all exercises (in particular, the programming problems), 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 of teaching hours, or on one of the InfoCommons computers (available in the university libraries and other teaching spaces).

Objectives

The purpose of this week's lab is to:

- understand the two different ways of interacting with python and which one to use;
- understand the evaluation of expressions in python, and something about the types of values;
- write functions that perform calculations.

Different ways of interacting with python

There are two different ways of interacting with python. You saw both of them in last week's lab, but it is worth talking about them a little more here so you understand the differences between them.

The first way to interact with python is to write code in the shell. You can tell you are working in the shell by the:

```
In [1]:
```

or

```
>>>
```

prompts at the beginning of each line. There are some differences depending on the development environment you are using (PyCharm, Spyder, IDLE, etc), but they all have some standard features:

- When you write a line of code, python (usually) executes it immediately.
- Python prints the return values of any expressions you evaluate in the shell, whether you ask it to or not.
- If you want to run something twice, you have to type it in twice (or use the up/down arrows to find it in your input history).
- You cannot keep anything between sessions (in other words, when you close the shell, your program is lost).
- Python will let you write functions in the shell, but if you want to change them later, you will have to re-type the entire function. This means that if you want to write any code that you want to test several times (such as, for example, a homework solution) then you should **not** do it in the shell.

The second way to interact with python is to write your code in a file, and then ask the python interpreter to execute all the code in the file in one go. This behaviour is more consistent between different development environments. It has the following features:

- Nothing gets executed until you actually run the code file.
- No output is printed unless you tell python to print it (or you trigger an error message).
- If you want to run the code a second time, you just hit the run button again.
- Python programs can be saved and stored just like any other file, and can be re-opened if you want to edit or use them again later.

What this means in practice is that while the shell is very useful for quickly checking or testing something, it isn't so good for larger, more complex programs or anything you want to keep around and use again later.

In the labs, we will often start off doing some testing in the shell so you can see how things work. However, further on in each lab there will be some larger problems that you should write in files so you can edit and change them (and re-run them). As the course goes on, less and less time will be spent in the shell, since it is a bit too limited for what we will be doing. However, don't forget that it's there if you need to quickly check something.

Exercise 0: Interactive evaluation

Start a python shell (by starting Spyder, IDLE, `ipython`, or `python3`). Try the following examples of simple arithmetic expressions. Make sure you understand what each of the arithmetic operators does, and

why you get the results you're seeing. If there is anything you don't understand, discuss with the person next to you or ask your tutor.

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

In [2]: 7 / 3
Out [2]: ...

In [3]: 7 // 3
Out [3]: ...

In [4]: 3 * 5
Out [4]: ...

In [5]: 10 ** 2
Out [5]: ...

In [6]: 4 / 2 * 5
Out [6]: ...

In [7]: 2 + 3 * 5
Out [7]: ...

In [8]: 3 * 5 ** 2
Out [8]: ...
```

(... indicates the response that the python interpeter gives you; don't type in the dots!)

If `7 / 3` gives you `2` instead of `2.33333333`, you are running *the wrong version of python* (`python2`, not `python3`). Quit the interpreter and start the right version!

`//` is the "floor division" (or integer division) operator, and `%` is the remainder (or "modulus") operator. As long as you use them with positive integer arguments, the results should be intuitively clear. However, they can also be applied to negative and fractional arguments, with sometimes not so intuitive results.

Exercise 1: Values (objects) and types

In python, every value computed by the interpreter is an *object* (although the word "object" has a very particular meaning). Every object has a *type*, which tells us (and the python interpreter) what kind of object it is.

The type of an object determines what operations can be performed on it, and what the result of those operations will be. For example, the multiplication operator (`*`) can be applied to two integers, and will result in another integer, but multiplication of two strings makes no sense.

An object can be associated with zero or more (variable) names. We can use these names to refer to the object.

Try the following and observe the results:

```
In [1]: type(2)
Out [1]: ...

In [2]: type(2.0)
Out [2]: ...

In [3]: type("2")
Out [3]: ...

In [4]: type("2.0")
Out [4]: ...
```

Type is an attribute of the value. Names (variables) in python do not have a type; that is, there is no restriction on the type of value they can be associated with, and and the type can change as the associated value changes during execution of a program. Therefore, it is not correct to talk about "the type of a variable"; we should say "the type of the current value of a variable".

(Many other programming languages, like, for example, C or Java, specify a type for each variable; the variable can then only have values of that type.)

Try the following:

```
In [1]: my_int = 13

In [2]: type(my_int)
Out [2]: ...

In [3]: my_float = 3.14

In [4]: type(my_float)
Out [4]: ...

In [5]: my_int = my_float

In [6]: type(my_int)
Out [6]: ...

In [7]: my_int
Out [7]: ...
```

If the results surprise you, you can try stepping through them using pythontutor.com.

Exercise 2: Floating-point numbers

Floating-point numbers represent decimal (fractional) numbers. As mentioned in the lecture, the floating-point number type has limited range and limited precision.

A consequence of the limited precision is that calculations on floating-point numbers are sometimes approximate. As an example, try the following:

```
In [1]: (11111113.0 + -11111111.0) + 7.51111111
Out [1]: ...

In [2]: 11111113.0 + (-11111111.0 + 7.51111111)
Out [2]: ...
```

You should find that the two sums evaluate to (slightly) different values, even though they are, mathematically speaking, equal. Here is another example:

```
In [3]: import math

In [4]: math.cos(math.pi / 2)
Out [4]: ...
```

The result may be close to zero, but not actually zero.

Another consequence of this imprecision is that you should normally not compare floating point numbers using equality. Try the following:

```
In [5]: 0.1 + 0.2 == 0.3
Out [5]: ...

In [6]: 0.1 + 0.2
Out [6]: ...
```

Remember that == is how we test whether two things are equal.

Instead, to check if two floating-point numbers are equal, check if they are "close enough", that is, if the difference between them is small enough. For example, the expression

```
math.fabs(x - y) < 1e-6
```

evaluates to `True` if the difference between `x` and `y` is less than 10-6. In general, you will need to pick a threshold that is precise enough for the purpose of the calculation you are implementing.

We also saw in the lecture that type `float` has two special values: `inf`, which represents infinity, and `nan` ("not a number") which is an error value that results from certain operations when the result is undefined (for example, `inf - inf`).

In recent versions of python, the `math` module provides a constant for infinity. Thus, we can get an infinite value as follows:

```
In [1]: import math

In [2]: x = math.inf

In [3]: x > 1e308
Out [3]: ...
```

However, this constant may not be available in older python versions. If it is not, the commands above will produce an `AttributeError` error message.

Exercise 3: Data analysis

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

According to the 2016 census data, the population of Canberra (the electoral division not the city) consists of:

- 196,037 people (adults and children).
- 50,352 families.

Of the families, 22,850 are a couple with one or more children and 7,243 are a single parent with one or more children. The rest are families without children, which consist of two (adult) people. The average number of children is 1.8 / family; this is the average over the families that have children only.

Write python code to calculate how many single adults (not children) there are in Canberra.

Start by assigning your initial data to appropriately named variables, for example:

```
canberra_population = 196037
canberra_families = 50352
```

Remember that you can not write large (integer) numbers with commas in python code!

If you calculate an intermediate result, it is good practice to also store it in a variable as well. For example:

```
people_per_household = canberra_population / canberra_families
```

Don't forget that when writing code in a file, if you want python to display information to the console, you must use the print function. For example, you can write

```
print("number of people per household:", people_per_household)
```

When you have written a program that calculates an answer, compare your answer with those of other students around you in the lab. Did you all get to the same answer? If you did not, ask them to take a look at your code: can they read your code and understand your method of calculation, without you explaining it? What comments would be useful to add to your file to help explain it?

Exercise 4: Functions that return a value

In Lab 1 you defined functions that combined primitive instructions to the simulated robot to make it accomplish bigger tasks. As shown in last week's lectures, functions can also encapsulate calculations, using parameters for the values that go into the calculation and the `return` statement to pass the result of the calculation back.

First, as a quick reminder, here is an example of how to define a function:

```
def odd(n):
    return 2 * n - 1
```

Write (or copy) the function definition into a file (you can call it `first_function.py`, for example) and run the file. You should see that nothing happens; this is because your code defines a function, but never calls (uses) it.

After you have run a python file in your IDE (or in the terminal using the `-i` option), the python shell will remain in interactive mode. That means you can test your function in the shell, by calling it and observing what it returns. For example:

```
In [1]: odd(1)
Out [1]: 1

In [2]: odd(2)
Out [3]: 3

In [3]: odd(10)
Out [3]: 19
```

Remember that the python shell will automatically print the value of an expression when in interactive mode, that is, when then expression is entered into the shell. If you add the tests above to your python file:

```
def odd(n):
    return 2 * n - 1

odd(1)
odd(2)
odd(10)
```

and run it, again you will see no output. To print the output, we have to add explicit calls to the print function:

```
def odd(n):
    return 2 * n - 1

print(odd(1))
print(odd(2))
print(odd(10))
```

Programming problems

The following problems ask you to write one or more functions to implement a calculation. Most of the examples are (somewhat) mathematical. However, none of them require you to solve a mathematical problem; the solution (calculation steps) is given, and what you have to do is translate that calculation into a working function in python.

Programming problem 4(a)

(Adapted from exercise 2-2.2 in Downey's book.) The price of a book is $24.95, but an online book seller is offering a 40% discount. The shipping cost is $3 for the first copy $0.75 for each additional copy.

Write a function, `total_price(n)`, that takes the number of copies ordered and returns the total price.

Testing your function Write your functions in a file called `total_price.py`. As shown above, you can run the file and then leave the python shell in interactive mode, so that you can test your function. For example,

```
In [1]: total_price(60)
Out [1]: ...
```

Remember the procedure for testing a fuction: First, specify the assumptions that you've made in writing the function, for example about the type and range of parameter values. In the case of the `total_price` function, for example, we expect of course that the argument is an integer. Then test the function with a wide range of values that fall within your assumptions.

- What test cases should you use for the `total_price` function?
- Does your function work for *any* number of copies?

Plotting a function

In the lectures, we have seen a few examples of how to plot a function. Here is a list of the steps (without detailed explanations):

1. Import the functions:

   ```
   In [1]: from matplotlib.pyplot import plot, show
   In [2]: from numpy import linspace
   ```

   If this causes an error, it is probably because you don't have the `matplotlib` library installed (or, if you are on a CSIT lab computer, because you started the wrong version of python).

   The statement above imports only a few functions from the modules, but these are all we will need right now.

2. Define the x-values:

```
In [3]: xs = linspace(-2.0, 4.0, 60)
```

This creates a sequence of values with even spacing, in this example 60 values evenly spaced between -2.0 and 4.0. You can of course use other ranges.

3. Compute the y-values:

```
In [4]: ys = [fun(x) for x in xs]
```

Here, `fun` is meant to be the function you want to plot (for example, `math.sin`, or any other function of one argument). This creates a sequence of values corresponding to your sequence of x-values, by calling the function once for every value in `xs`.

If you want to see what the numbers in the sequence are, just type the variable's name into the shell:

```
In [5]: xs
Out [5]: ...
In [6]: ys
Out [6]: ...
```

4. Plot the values:

```
In [7]: plot(xs, ys, linestyle="solid", color="blue")
In [8]: show()
```

In the `ipython` shell (which is the default shell in Spyder), the plot will show in the shell console. If you are using the standard `python3` shell, or IDLE, it will open in a new window, and the `show` function will not return until you close that window. You can make several calls to `plot` before calling `show()`, if you want to see several lines in the same figure.

Programming problem 4(b)

The Voyager 1 spacecraft, launched on the 15th of September in 1977, is the farthest-traveling Earth-made object. On the 25th of August in 2012, it crossed the heliopause, which can be considered the boundary of the solar system. According to the JPL voyager web site, Voyager 1 is currently (31st of July 2019) approximately 21,906,219,000 kilometers from the Sun, traveling away from the Sun at approximately 16.9995 km/second.

JPL lists the current round-trip time for radio communication as 40 hours and 27 minutes (145620 seconds). Radio waves travel at the speed of light, which is approximately 299,792,458 meters/second.

Write a function that calculates the round-trip communication time at a future date. Your function should take one argument, which is a number of days after today, and return the round-trip time in seconds. (You don't need to adjust for the additional distance travelled during time that the radio transmission is in flight.) For reference, the equations are:

distance = distance at start + velocity * time since start

round-trip communication time = 2 * distance / speed of light

You will probably find it easier to break the calculation up into steps, such as first calculating the distance, then the radio signal travel time.

Use your function to estimate the round-trip communication time 1, 3, 10, 100 or 300 years from now.

Your calculated time will probably differ from that given by JPL, because they are calculating the round-trip communication time between Voyager 1 and the Earth. The Earth's average distance from the Sun is 149.598 million km (which is also known as one Astronomical Unit, or AU), so the distance between Voyager 1 and the Earth may differ from that between Voyager 1 and the Sun by plus or minus 1 AU. To check that your answers are reasonable, work out how big a difference this can make to the round-trip radio communication time, and check if the difference you get is less.

(Note: There is a similar exercise in Punch & Enbody's book, programming problem 1 on page 79; however, the values they give for Voyager 1's distance and speed do not agree with JPL's.)

Programming problem 4(c)

This and the following problem are a bit more mathematical in nature. You can skip these two problems if you find that the mathematical content, rather than the actual programming, is too difficult.

A linear equation in one unknown is an equation of the form

$$ax = b$$

where

$$a$$

and

$$b$$

are constants (numbers) and

$$x$$

is the unknown variable. This is easy to solve: the answer is

$$x = b/a$$

(and it has no unique solution if

$$a = 0$$

).

A system of two linear equations in two unknowns looks like this:

$$a_1x + b_1y == c_1$$

$$a_2x + b_2y == c_2$$

where

$$a_1$$

,

$$a_2$$

,

$$b_1$$

,

$$b_2$$

,

$$c_1$$

and

$$c_2$$

are constants, and

$$x$$

and

$$y$$

are the unknowns. There are several methods for solving such a system of equations, but the one we'll try first is by substitution. First, find the ratio

$$r = a_1/a_2$$

. It follows that

$$(b_1 - rb_2)y = c_1 - rc_2$$

This is a single equation in one unknown, which we know how to solve. That gives us the value of

$$y$$

, and we can now reorganise one of the original equations into

$$a_1x = c_1 - b_1y$$

Here the value of

$$y$$

is known, so the right-hand side is just an arithmetic expression that can be evaluated. This again brings us back to a single equation in one unknown, which we know how to solve.

- Write a function `solve1(a, b)`, which solves the linear equation

  $$ax = b$$

  and returns the value of

  $$x$$

  .

- Write a function `solve2(a1, b1, c1, a2, b2, c2)`, which solves the system of two linear equations shown above. To return the values of both

  $$x$$

  and

  $$y$$

  , you can write a statement as follows:

  ```
  return x, y
  ```

  (assuming `x` and `y` are variables in your function) and you should see when you call the function that you get a pair of two values. (It is actually a single value, which is a sequence of two values; we will see much more about sequences later in the course.)

- Does your `solve2` function call `solve1`?

- Can you generalise the substitution method to a system of three equations in three unknowns? (and write a function `solve3` that returns all three values).

Testing your function Write your functions in a file called `equations.py`. When you run the file in Spyder or IDLE (or in the terminal using the `-i` option), the python shell will remain in interactive mode after finishing the program. You can then test your function in the shell. For example,

```
In [1]: solve1(1, 2)
Out [1]: 2.0

In [2]: solve1(2, 1)
Out [2]: 0.5

In [3]: solve2(1, 1, 1, 2, -1, 0)
Out [3]: (0.3333333333333333, 0.6666666666666666)
```

Programming problem 4(d)

Another method of solving a system of two equations is using Cramer's rule, which expresses the solution in terms of the determinants of the coefficient matrices. (This may be familiar to Engineering students, since Cramer's rule is covered in ENGN1217 and ENGN1218.) First, state the equations in matrix form:

$$\begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}$$

The determinant of a

$$2 \times 2$$

matrix,

$$\begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \end{bmatrix}$$

, is

$$a_1 b_2 - b_1 a_2$$

. Cramer's rule states that

$$x = \frac{det(\begin{bmatrix} c_1 & b_1 \\ c_2 & b_2 \end{bmatrix})}{det(\begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \end{bmatrix})}$$

$$y = \frac{det(\begin{bmatrix} a_1 & c_1 \\ a_2 & c_2 \end{bmatrix})}{det(\begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \end{bmatrix})}$$

where `det` stands for a function that computes the determinant of the matrix.

- Write the function that computes (and returns) the determinant of a

  $$2 \times 2$$

  matrix.
- Use this function to implement function for solving a system of two linear equations using Cramer's rule.

Now that you have implemented two methods of solving a system of two linear equations, you can test your functions by comparing the results of both methods on each test case.