

Semester 2, 2018: Lab 1

S2 2018

Lab 1

This lab has two parts. Part A is intended to help you familiarise yourself with the computing environment found on the CSIT lab computers which you will be using throughout the course. If something unexpected happens, or if you get stuck, you can either ask another student or your tutor. Part B introduces your first programming exercises.

Both parts should be completed during semester week 2. It is not expected that you will be able to finish all the programming exercises during the scheduled 2 hour lab session. The normal workload for a 6-unit course is a bit over 10 hours per week. That means students in this course are expected to spend at least 5 hours per week practicing programming in addition to the scheduled lecture and lab times.

You should have 24/7 access to some of the CSIT labs (N112, N113 and N114) - just use your student card to swipe in - so that you can work there whenever there is no scheduled activity (usually after 7pm on weekdays, and on weekends). Sometimes lab rooms will not be full even when there is a scheduled class, in which case you can often take a spare place. However, you should always check with the teacher in the ongoing class that it is ok, and you should of course not disturb students in that class. The teacher always has the right to ask you to leave a computer lab room when there is a scheduled class that you are not enrolled in, regardless of whether it is full or not.

Anaconda python is also installed on the ANU InfoCommons computers. These are the Windows and Mac computers found in, for example, libraries and lecture theatres across the campus. This means you can work on programming exercises also on those machines.

Part A

For these exercises, you will need to be sitting in front of a computer in the CSIT labs.

The main objective of this part is to ensure that you have a working computer account, are able to use some of the utilities of the CSIT student computing environment, and can run basic python programs on this system. Part B consists of a set of programming exercises. From next week on, we will focus solely on python programming.

Background

The computers in the CSIT labs are running the GNU Linux operating system (Ubuntu and XFCE). If you have never used a Linux system before, don't panic! The basic concepts you will be working with (files, directories, applications, etc) are similar to those in other operating systems such as Microsoft Windows and Mac OS X. (In fact, if you've used OS X, you have used a system that is very similar to Linux behind the scenes.) There is a bit of a learning curve, but it shouldn't be long until you feel comfortable.

You do need to be aware that the CSIT computing environment is not the same as that found on other computers at the ANU. You should now be able to access your ANU-wide home drive (file storage) - look for an application called ANUHomeDrive:

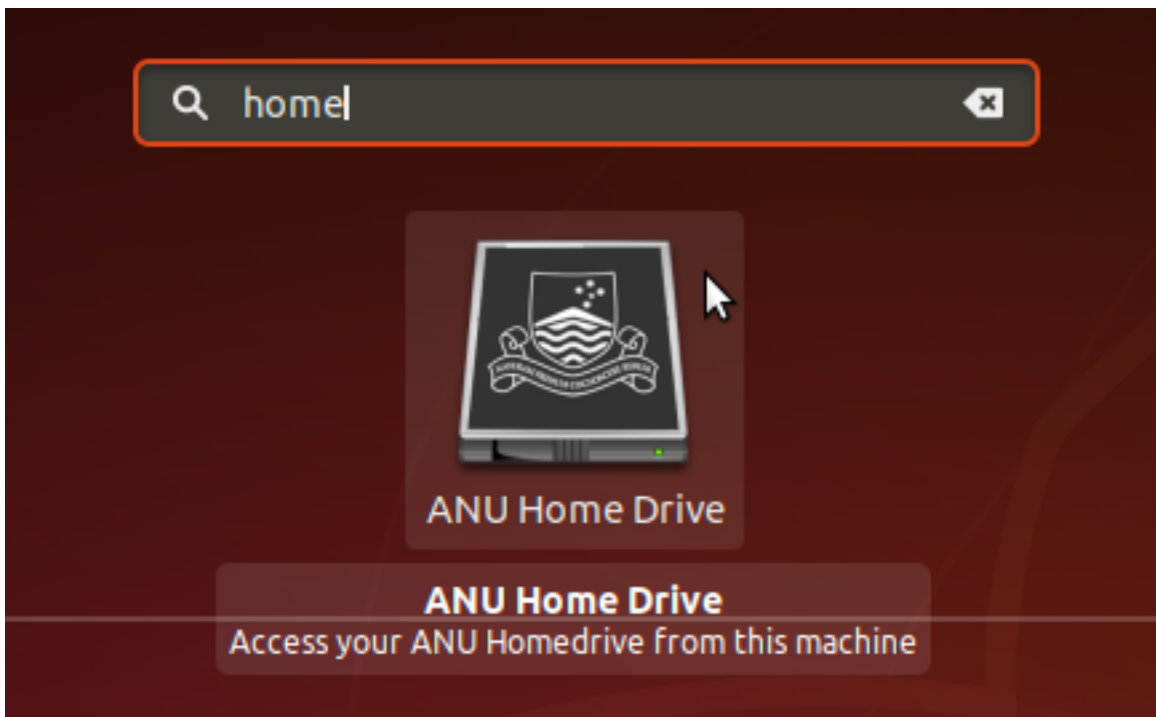


Figure 1: ANU Home Drive icon

Files that you store here will be accessible from other computers on campus, such as the InfoCommons Windows and Mac computers. Other ways you can move your files around is by copying them onto an external memory drive (e.g. USB stick), through an on-line service such as dropbox, or emailing them to yourself.

Exercise 0: Logging in and out, locking the screen

You log into the CSIT computers using your student identification number (prefixed by a "u") and the same password as elsewhere in the university. *IF*, however, you have never logged into the CSIT systems

before then you need to first log on to [STREAMS](#). This will trigger creation of an account on the CSIT system. This was mentioned in the first lecture so hopefully everyone has done this. If you have not, then you will need to do this during the lab using a web browser. The creation of the account is not instantaneous. It can take up to a day to take effect.

If you have not created your account in advance of the lab, you can log in on the CSIT lab computer using a “guest session”. This will let you use programs on the computer (such as the web browser, python interpreter, and so on). However, if you are logged in as guest, you will not be able to save any work (except using external memory, or an on-line service): all files that you save **will be automatically deleted when you end the guest session**.

It is important that when you finish with the computer you log off. This is done by clicking on the icon in the top right hand corner that looks like an on/off button. This will bring up a menu with one option that is “log off”. Try logging off and on again now.

If you step away from your computer for a short period then you can lock the screen. This is another menu option you should try. Recognise that if you lock the screen and go away for a long period of time another person is likely to come and reboot the machine - this will log you off and could result in you losing any work that you have not saved. So before locking the screen, save all your work. If in doubt about what this means - ask a fellow student or your tutor.

Never leave the computer without either logging out or locking the screen! If you leave the computer unlocked, other people can read (and change or delete) everything that is on your account. They can also do malicious things to other people under your identity. You are responsible for anything done on, or through, a computer logged in with your user name.

Exercise 1: Exploring the desktop

Much like Windows and OS X, Linux provides a graphical user interface, which follows a consistent set of guidelines. You will find windows, menus, control panels, consistent visual user feedback, direct manipulation and interaction between programs, and other aspects of modern graphical user interfaces.

When you first log in, you will see a (mostly) empty desktop. Depending on whether you chose “Ubuntu” or “XFCE” as your desktop at login, you will have one of two different views: The way that you find and start application programs is slightly different between the two.

In XFCE, programs are found under the Applications menu in the top panel. It has several sub-menus: For example, the Firefox web browser is found under the “Internet” sub-menu. The tools you will need for writing and running python programs are mostly found under the “Development” sub-menu:

In Ubuntu, a few programs are accessible by buttons on the panel on the left side of the desktop. For other applications, click the button at the bottom of the panel (or the menu key on the keyboard), type in a few letters from the beginning of a word in the name of the application you’re looking for and icons for matching applications should appear. Then click on the one you want:

Locate and start the following

- a web browser (for example, Firefox)
- Anaconda Navigator

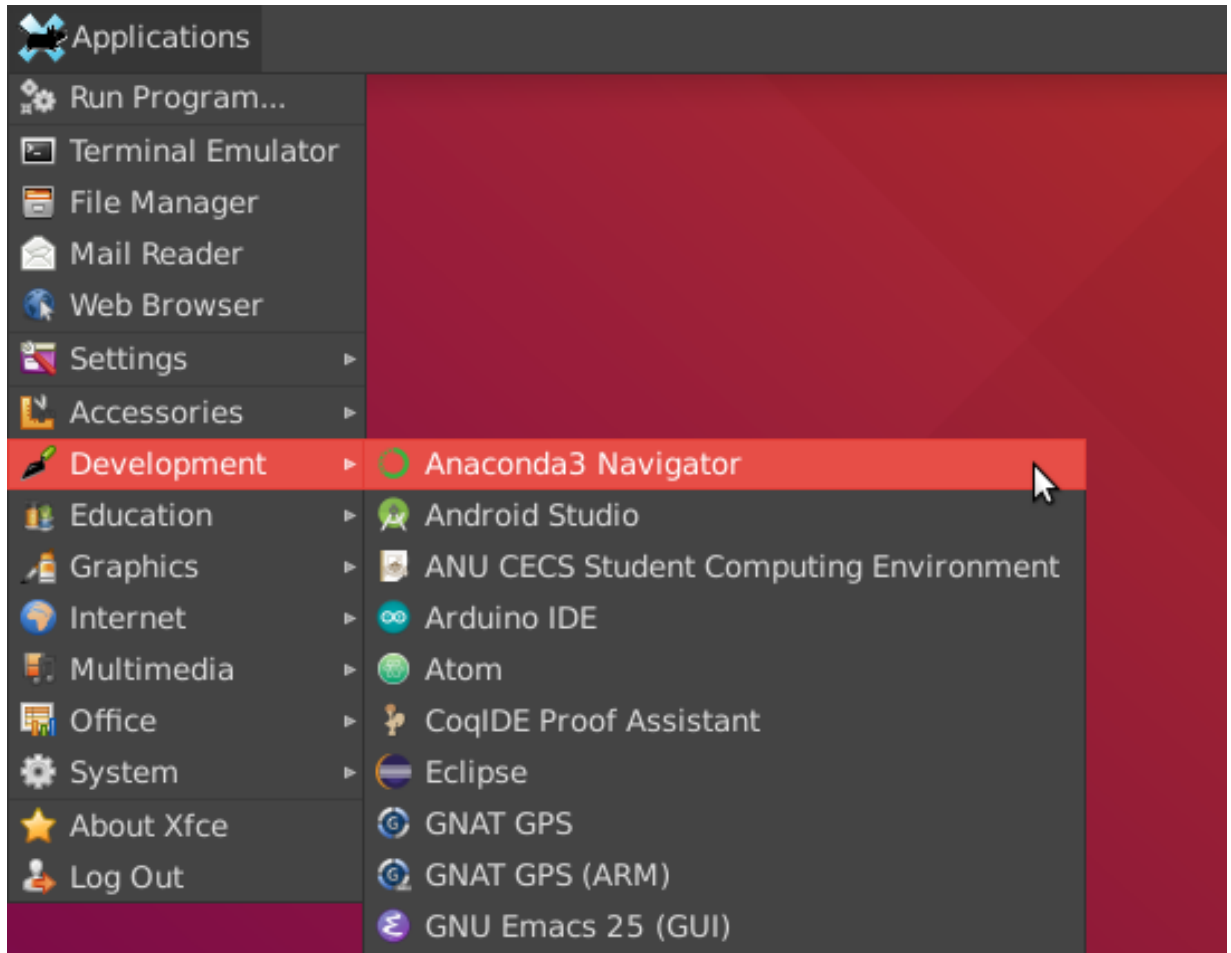


Figure 2: screenshot XFCE menu



Figure 3: screenshot Ubuntu menu

As you become more experienced, you may want to customize your desktop to better suite your personal style.

Exercise 2: Files and folders

Select “Home Folder” from the “Places” menu (GNOME), or the folder icon from the sidebar (Ubuntu).

This will open a window displaying the contents of your home folder. You can create files and folders using this graphical interface.

First, in your home directory, create a new directory (folder) named “`comp1730`”. In the `comp1730` folder, create another named “`lab1`”.

Download and save this small python program: [print_brick_wall_1.py](#). Assuming you are using the Firefox browser, you can do this by right-clicking on the link and selecting “Save link as...”. (The way to do it in other browsers may be slightly different, but all should have the capability.) Save this file in the `lab1` directory that you created.

Part B

Choosing an IDE

An *Integrated Development Environment*, or IDE, for python combines a text editor (for writing programs) with a python shell (interpreter), and some helpful functions to integrate the two.

Different IDEs are available in the CSIT lab and the InfoCommons enviroments. You will need to learn to use one of them. (You can also use a stand-alone text editor, and the `python3` or `ipython` interpreter through the command-line terminal.) If you have no strong preference, we recommend that you use the Anaconda Spyder IDE, since it’s relatively easy to use and available on all platforms.

The **python shell** (or **interpreter**) is the program that executes python programs and interactive commands. There are two different shells: the standard **python3** shell and **ipython**. Under the hood they are mostly the same, but they look a little different. When you start the **python3** shell you should see a message like this

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:09:38)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The characteristic prompt `>>>` indicates that the interpreter is waiting for you to type in a command or expression. (The Python version number is somewhat different in different environments. For example, on the InfoCommons computers it should be `Python 3.5.2`. The important thing is that it is *python 3*.) In **ipython**, the prompt will instead look like this


```
In [1]:
```

where **In** means it is waiting for your input. (The number in brackets is a running counter of inputs and outputs.) Examples on these lab pages will be written using the **ipython** shell prompt. There are a few other differences between the two shells: **ipython** gives you some shortcuts to make typing in commands interactively easier, and prefixes the values that results from evaluating an expression with **Out**. When you use certain types of graphics (such as function plots), **ipython** will show them inline in the shell window, while the **python3** shell will show them in a separate window.

Spyder is the IDE that comes with Anaconda. It is available in both the CSIT lab environment and on the InfoCommons computers. To start Spyder, launch the *Anaconda Navigator*, which you can find in the start menu on the GNOME and Ubuntu desktops, and in the Windows start menu. From it, just click “Launch” below the Spyder app to start the IDE.

The Spyder IDE provides you with an editor (to the left in the image below), a python interpreter (below on the right) as well as a window for help messages, variable inspector, and other things:

Spyder by default opens both a standard **python3** and an **ipython** shell. (You switch between them using the tabs at the bottom, highlighted in the image above.) This can be a bit confusing, since things you do in one of them do not affect the other. It’s best to just close the one you don’t want to use.

To run a program that you’re editing or writing in Spyder, click the run icon (). The first time you run a file this will bring up a dialog with some options: just click “Run” to accept the defaults.

IDLE is the default IDE for python, and comes with most python distributions. It is more simplistic (some would say it is just bad), but many students will find it sufficient for the purpose of this course. You can find “Anaconda IDLE” in the start menu on the GNOME and Ubuntu desktops. On the InfoCommons Windows or Mac computers you will have to use the search function on the start menu and search for “IDLE” or “python IDLE”. There is more than one version available, so check after you have started it that you have the right one (it should show a message like the one above). In particular, make sure it is running python 3.

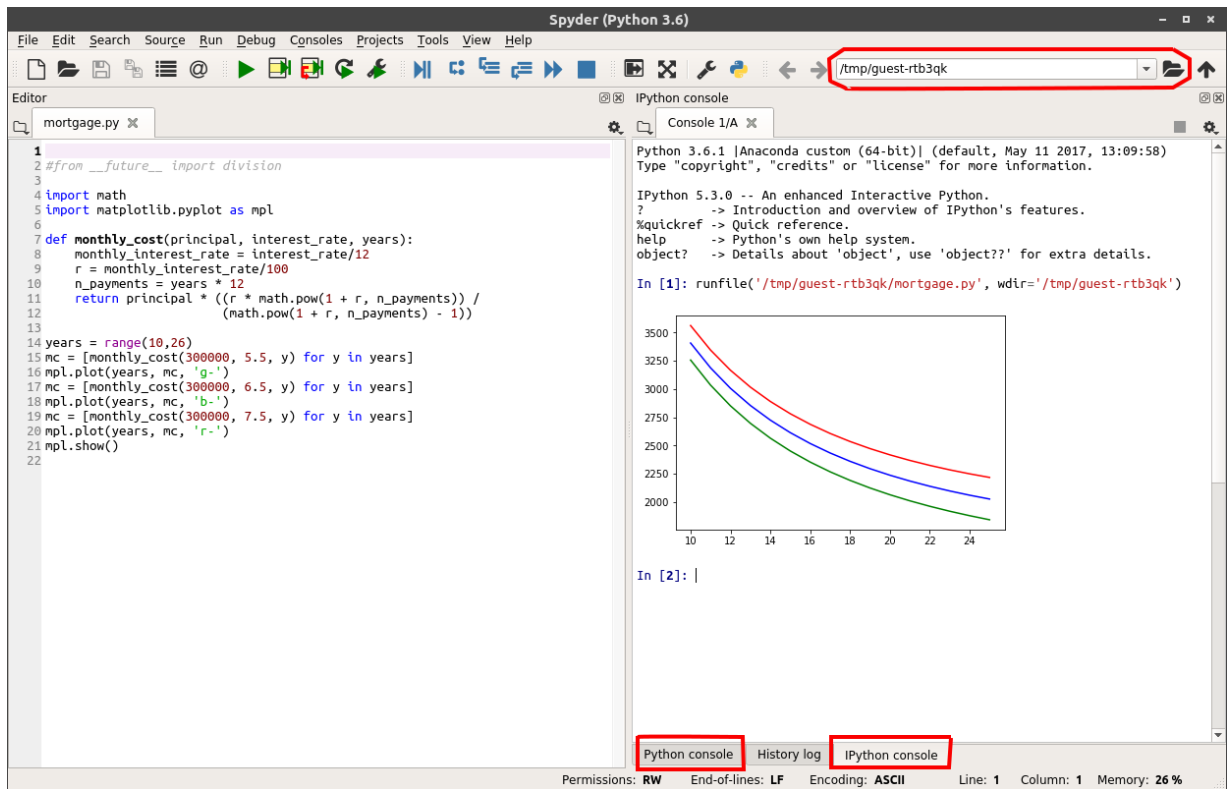


Figure 4: Spyder IDE

IDLE has a separate window for the python shell, and one window for each file you edit. To run a program in IDLE, you can select “Run Module” from the menu in the editor window, or press F5.

The **PyCharm** IDE is available in the CSIT lab environment. You can find it in the “Programming” sub-menu on the GNOME desktop, or by searching in the Ubuntu start menu. Like Spyder, it provides a split-window layout, with an editor, a python shell, and other tools. Note that PyCharm does not open a python shell on start-up, but only when you run a program. Before you can run a program in PyCharm, you will need to set the python interpreter. Click on the “Configure Python Interpreter” link:

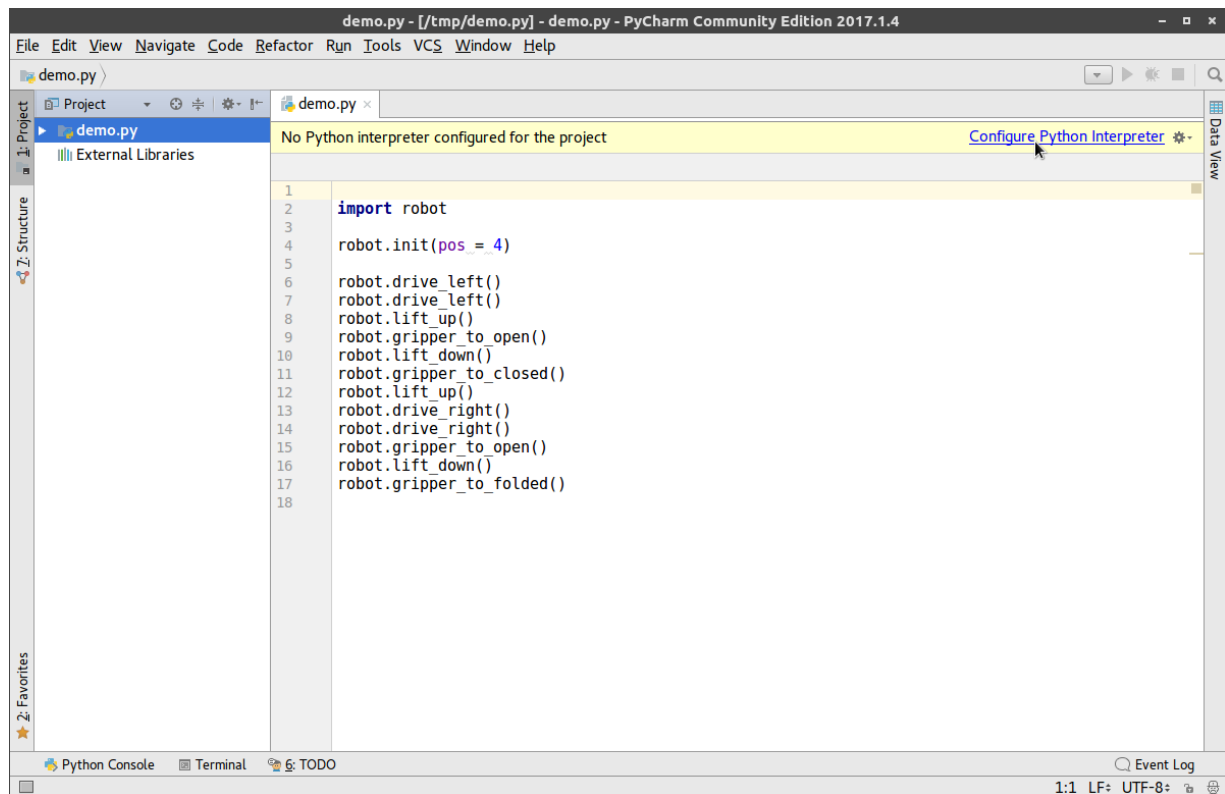


Figure 5: PyCharm, with configure link

and select (or type in if it’s not available as a choice) `/usr/local/anaconda3/bin/python3`.

You will see us making use of PyCharm in the lectures a lot.

Exercise 3: Debugging

Open the file `print_brick_wall_1.py` that you saved in Exercise 2, and attempt to run it. The program has several syntax errors, so it will not run. (Depending on which IDE you are using, you may even see some errors highlighted before you try to run the program.)

Identify and correct the syntax errors so that the program runs. The aim of this simple program is to print a “brick wall” pattern on the screen, like this:

```
--+-----+---  
  |         |  
-----+-----+  
      |     |  
--+-----+---  
  |         |  
-----+-----+  
      |     |  
--+-----+---  
  |         |  
-----+-----+  
      |     |
```

If you cannot find or fix the errors, ask your tutor or another student for help. In case you get stuck, a “cheat sheet” is provided [here](#).

However, even after the syntax errors have been corrected, the program may not print what it should (i.e., exactly the output that is shown above). Modify the program so that you get the correct output. Do not write a whole new program! You should understand how this program works, and fix it so that it does the right thing.

Exercise 4: Programming the robot simulator

Download a copy of the [robot.py](#) module and save it in your `lab1` directory.

Before you can do any robot programming, you have to make sure that you can import this module. Remember that python will look for the module in the *current working directory* (“`cwd`”).

When you run a python program, the `cwd` is the directory that the program is in. Thus, an easy way to make `import` work is to create a new program, call it `first_test.py`, and save that too in the `lab1` directory.

Note: All your python programs must be saved with a name that ends in `.py`.

Your program should contain the single statement

```
import robot
```

Run the program. After this, you should be able to run robot simulator commands in the shell. Remember that you must start by initialising the simulation:

```
In [1]: robot.init()
```

After this, you can test driving the robot around:

```
In [2]: robot.drive_right()
In [3]: robot.lift_up()
In [4]: robot.gripper_to_open()
In [5]: robot.lift_down()
```

The python help system

Python has a built-in `help` function, which gives you access to documentation in the python shell. Try it out on the robot module:

```
In [6]: help(robot)
...
```

Programming problems

1. The default simulator setup (what you get when you start the simulator with just `robot.init()`) has three boxes on the shelf. It also has a limit of one on the height that the robot can reach. (Height zero is on the table; height one is one step above.)

Write procedures (functions) to make each pair of the boxes swap places, i.e., one to swap the left and middle box, one to swap the middle and right box, and one to swap the left and right boxes (ending with the middle box in the middle). Recall that a function definition is done like this:

```
def swap_left_and_middle():
    ...function suite..._
```

The function suite is a sequence of statements. The extent of the suite is defined by indentation (whitespace before the statement): all statements in the suite must have the same amount of indentation. The standard is 4 spaces or 1 tab.

Remember to identify the assumptions of your function. Do you assume that the robot is standing in front of the left box or the middle one at the beginning? What is the assumed state of the lift and gripper? Document your assumptions in code comments.

You will likely find that defining functions in the shell is quite frustrating. Put your function definitions in `first_test.py` which you created earlier. Then, after you have written each function definition, run `first_test.py`, the same way you ran `print_brick_wall_1.py`. You should now be able to call the new function from the shell.

2. Can you identify some manoeuvres that are common to all three problems? Split those off into separate functions. Your code should become more compact and easier to read.
3. The simulator allows the robot to lift a stack of boxes, not just a single box. However, if we try to do this with the real robot, chances are the stack will fall over. Try to write your functions so that the robot accomplishes each swap without ever lifting more than one box at a time.

Exercise 5 (finding and using modules)

Python comes with over 200 modules in the standard library, and many more optional modules can be installed. If you type

```
In [1]: help()
```

at the prompt, you enter python's built-in help system. The prompt will change to

```
help>
```

Type `quit` (and press Enter) to leave the help system and return to the normal python shell.

You can get information about a module by typing in the module's name, for example:

```
help> math
```

This will print the complete listing of all functions, variables, classes, etc, that the module defines. This may be a bit more than what you're ready for at this point, so let's look at some other ways to find information.

First, most python shells support tab completion. This means that if you type in the beginning of name and press the Tab key, the shell will display a list of defined names that match that beginning. (IDLE will show the list as a popup menu.) This works with modules too, but only if they have been imported. For example, if you type

```
In [1]: import math
In [2]: math.
```

and then press the Tab key, you should get a list of all names defined in the `math` module. From the python shell, you can also use the `help` function to get information about specific names, such as,

```
In [3]: help(math.exp1)
```

Note that here you did not enter into help mode.

Another way to find information is on the web. Open a browser and go to [python docs](#). (By default, the site will show you the documentation for the latest version of python3. If you are looking for documentation for another version, select it from the side bar, or using the menu in the top left corner.) Follow the link "Library Reference": this will take you to the main index for the python standard library. Find the module you're interested in on the page, and open its documentation page.

You can also find a lot of useful help just by typing a query into a search engine (such as any one of the [40 active search engines listed on this wikipedia page](#)). Make sure you include the name of the programming language - python. You may also find it that you get more useful answers if you include the version number - 3.X - and specific details about your query.

Problem

Find a module that has a function that gives you the current date. Use it to print a message, such as:

```
In [1]: print("Today's date is:", X)
```

(replacing X with a call to the function that you found).

The `print` function prints text to the terminal window. You can give it several arguments, which will all be printed in sequence, on a single line. For example,

```
In [2]: print("The sum of ", 2, "and", 3, "is", 2 + 3)
```

Note that bits of text are always enclosed in quotation marks.