Lab 01: Setup and the Basics

Contents

Objectives

By the end of this lab you will understand:

- What pair programming is
- Basic types in Python
- How labs will be run in this module
- Best practices for commenting and testing code

By the end of this lab you will be able to:

- Navigate around an operating system
- Interact with git repositories
- Use a programming environment
- Use Python as a calculator

1 Overview of Labs in SECU2002

Welcome to your very first SECU2002 lab! Over the next two hours, you'll learn some basic interactions with your computer's file system and with Python. You'll also learn how labs work in this module.

Each lab will start the same way: you'll access the `readme.pdf` for the week, which will act as your guide through the work you're expected to do. Because you are meant to work collaboratively during the labs, this document won't be available ahead of time. With the exception of the first few tasks in this lab, all lab work will be done with a partner.

Within each lab, you'll be given some information to augment what we've covered in lectures and you'll also be asked to do some *tasks*. You can do everything at your own pace, but occasionally you'll reach a checkpoint, which is clearly marked in the file. At this point, please call one of us over to quickly look over your work and ensure that you're on track.

In the beginning, labs will have many tasks, as you learn and experiment with the many features of Python. As you become more familiar with the language, you'll notice that the tasks will become fewer but will require significantly more effort in terms of problem solving. These are designed to challenge you, so please don't become discouraged if you find them difficult, and if at any point you and your partner feel really stuck you're always welcome to ask one of us for help.

In terms of marking, you are expected to hand in your work at the end of every lab; instructions for how to do so are at the end of this document (and will be there in all subsequent weeks). We will notice if you modify it after the end of the lab, so please don't try! Marking is out of 5 for each lab; the idea is that you can get 1 mark just for showing up, 4 marks if you put in a good effort but don't manage to solve everything, and 5 marks if you cover all edge cases and write nice code. You should get your mark back by the end of the day, and at the very latest by the next week's lab.

2 The Terminal

Using the instructions in `admin/setup.pdf`, you should already have the ability to access a terminal, which provides a direct way to interact with your computer's operating system, and in particular the file system. If you have not already gone through this file, please do so now before continuing with the rest of the lab.

Common commands in the terminal are `ls`, which returns a list of all the files in the current directory; `cd`, which changes the current directory (to the home directory if none is specified, and otherwise to the specified directory); and `mkdir`, which creates a new directory. In most file systems, files are organized hierarchically, with files contained within directories, and directories (also known as folders) located one within another. In Unix/Linux, the directory that contains all other directories is known as the *root* directory and is called `/`. The *path* to any other directory then starts at the root; for example, the home directory on Mac OS X is `/Users/<username>` for a user `username`, meaning the directory `<username>` is contained within the directory `Users`, which is itself contained within the root directory.

Task 1. Open the terminal. Identify the path to the current directory by typing `pwd` ('print working directory') and pressing Enter. Identify the contents of the current directory by running the `ls` command.

Because absolute paths are often quite long, the commands `.` and `..` serve as respective shortcuts for the current directory and the directory in which it is contained (the *parent* directory). It is also possible to navigate a file system using *relative* paths.

Task 2. Navigate to the parent directory of your current directory by running the `cd ..` command. Then, navigate back into the directory by running the `cd <mydir>` command, where `<mydir>` is just the name (not the absolute path) of the directory. If you do not know the name of the directory, run `ls` in the parent directory to identify it (or recall the end of the path from when you ran `pwd`). If you run the `cd` command without any argument, it will return to the home directory.

In addition to navigating around the file system, you can also use the terminal to create and delete files and directories.

Task 3. Create a new directory by running the `mkdir mydir` command (or feel free to use another directory name instead of `mydir`). Navigate in and out of this directory by running the same commands as before, until you're back in the directory where you started. Delete the created directory by running the `rm -rf mydir` command.

While `rm -rf` (or `rm -r`, or `rmdir`) is used for directories, `rm` suffices to delete files. *Please be very careful*, however, when using this type of deletion: unlike dragging the icon of a file or directory to the bin, as you may be used to, it is not possible to recover a file or directory once it has been deleted using `rm`.

3 Version Control

To get you started with the good practice of using Git early in your programming career, all lab material in this module will be maintained using Github, and all lab work will be handed in using Github. You should already have Git installed using the instructions in `admin/setup.pdf`, but please revisit that guide now if you don't.

Task 4. Create an account with Github, or use an existing one, and set up a new private repository called `secu2002_<uid>`, where `<uid>` is your six-character student number; you should be able to create unlimited private repositories if you register with an academic email address. Invite me (`smeiklej`) and Enrico (`EMariconti`) to join the repository as collaborators.

Create a local copy of your repository on your computer by creating a directory for it (using something like `mkdir secu2002_<uid>`) and then running

```
git clone https://github.com/<username>/secu2002_<uid>.git secu2002_<uid>
```

Inside of this directory, run `mkdir lab01` to create a directory for this lab, and put all of the files you create today into this directory.

The main command-line options that are possible in Git are as follows:

- Via `git clone <url> <dir>`, you can copy the contents of a repository maintained at `<url>` into your local computer's directory `<dir>` (as you did in the task above).
- As the contents of the remote repository are updated, you can update your local copy using `git pull` (or `git pull origin master`, depending on your configuration).
- If you make local changes to a repository, you must first *commit* these changes locally, using a command like `git -a -m "<commit message>"` (where `-a` adds all modified files, and `-m` adds the possibility of a log message). If this doesn't work for you, you may need to manually add the files by running `git add <filename>` and then run `git -m "<commit message>"`.
- You can then run `git push` (or, again, `git push origin master`, depending on your configuration) to push the local changes to the remote repository.

There are other useful Git commands, like `git status` (which will show you which files in your local copy have been changed), and `git log` (which will show you all previous changes by all collaborators), but for the purposes of this module we don't expect you to have to use these.

Task 5. You can see the git repository for this module at `github.com/smeiklej/secu2002_2017`. Create a local copy of this repository for the module by running `mkdir secu2002_master` inside of a relevant directory and then

```
git clone https://github.com/smeiklej/secu2002_2017.git secu2002_master
```

Checkpoint 1. You have reached a checkpoint! Please call the PGTA or lecturer over to check what you've done so far.

4 Pair Programming

Task 6. Read the pair programming guide available at `admin/pair.pdf` in the repository, and discuss it briefly with a neighboring student. Then, pick a partner to work with for the rest of this week's lab.

5 Numbers

Now that we have a solid foundation in place, we're ready to start writing code. There are two ways to write code in Python: using the interactive interpreter, which executes commands after they are entered and prints the result, and using files with a `.py` extension, which can then be run by entering `python <filename>.py` in the terminal (or by clicking the 'run' button in an IDE).

We'll write Python files later in this lab, but to start playing around with Python, let's begin with the interactive interpreter. This can be opened by entering `python` into the terminal, or by opening the 'Python Console' in PyCharm (View > Tool Windows > Python Console). In our code snippets, we use `>>>` to denote the prompt in the interactive interpreter.

5.1 The very basics

Task 7. Open the interpreter, and add, subtract, and multiply relatively small numbers using the `+`, `-`, and `*` operators. For example, if you type `4 + 1` and press Enter, you should see `5` appear on the line below.

5.2 Big numbers

So far, the interpreter should behave just as a normal calculator would. Now, let's see what happens if we use bigger numbers.

Task 8. Mash down on the numeric keys for a few seconds to generate a big number, and then multiply by another big number. Do this until the resulting number ends with an `L`.

What does this `L` mean? You've encountered your first *type*: a long integer `long`. (Actually, the first type was an integer `int`.) In Python3, these types have been merged.

5.3 Division and floats

Now, let's fold division into our calculator, using the `/` operator.

Task 9. Divide numbers by something that you know divides them evenly, like `4 / 2` and `10 / 5`. All good? Now try something else, like `1 / 2` or `10 / 4`.

Did you get an answer other than the one you were expecting? This is because of another issue involving types. If both of the values used in the division are of type `int`, then the return type is constrained to be `int` as well, which means what is returned is just something known as the *quotient*. (If you're curious, for all integers a and b there are unique integers q and r such that $a = qb + r$, where $0 \leq r < b$. We call q the quotient and r the remainder. To get the remainder, you can run `a % b`; e.g., `10 % 4` yields `2` because $10 = 2 \times 4 + 2$.) To get the 'real' result of the division, we must introduce the type `float`, which is a representation of a real number (i.e., a number in $\mathbb{R}$) and thus allows us to capture fractions.

One way to do this is to explicitly *cast* one or both of the numbers as a float, by running something like `float(4)`. Another way is to use its more precise representation like `4.0`.

Task 10. Using one of the above techniques, divide the same numbers that failed to produce the right result earlier, and ensure that you can now obtain it by incorporating `float`. Try many combinations; e.g., `10.0 / 4`, `10.0 / 4.0`, `float(10) / 4`, etc.

As with `long`, this is not an issue in Python3, which automatically *casts* the numbers in a division as `float`.

As mentioned in the lecture, certain operations can lead to *errors* in Python, in which the expression that you've entered cannot be evaluated for some reason. Conveniently, the error message you get provides information on the reason. For example, if you try to divide a number by zero (an operation that is not defined mathematically), you'll see the following:

```
>>> 10.0 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: float division by zero
```

You'll learn later on how to *handle* these types of errors, but for now simply take note of the type if you encounter them and try to understand the reason.

5.4 Exponentiation

Task 11. Try to square or otherwise raise a number to some exponent using the operator you would expect.

Did you use `^` and get a result other than the one you were expecting? This is because in Python (and most programming languages), `^` is not the exponentiation operator, but a logical binary operator known as *XOR* (short for exclusive or). Instead, the exponentiation operator is `**`.

Task 12. Raise a number to an exponent using `**`, and ensure that you get the results you were expecting in the previous task.

5.5 Order of operations

Does anyone remember PEMDAS from school? Or BOMDAS? These mnemonics tell us the order in which numerical operations occur (Parentheses, Exponentiation, Multiplication/Division, and Addition/Subtraction), and these rules are followed by all programming languages.

Task 13. Evaluate an expression mixing multiple operations without using parentheses and see what happens. Now, add parentheses to see how it affects the order of operations, and thus the outcome.

6 Types

It can be difficult to keep track of all the types within Python, and sometimes certain operations work only on certain types. In case you ever need to test the type of an object, there are two built-in functions to do this: `type` and `isinstance`. The first, `type`, will return the type of its input. The second, `isinstance`, takes in an object and a type and returns `True` if the object has that type and `False` if not. The output of this function is a new type called `bool` (which is short for Boolean, and of which `True` and `False` are the only instances).

Task 14. Test the types of a few different objects, using both `type` and `isinstance`. Try to cover all the types we have learned so far (`int`, `long`, `float`, and `bool`).

7 Assignment and variables

So far we have explored specific numeric *expressions*, which as we discussed in the lecture are executed in order to obtain a value. We now consider how to *assign* values to a *variable*, which allows us to reuse and update the underlying values many times throughout a program without having to manage it manually. To assign a value to a variable, we run a command like `x = 5`; the value on the right-hand side of the assignment statement is an expression that (when evaluated) takes on a specific value, and the value on the left-hand side is a variable. Running the statement has the effect of *binding* the variable to the value.

Task 15. Assign a single numeric value to a variable, and recall the value of that variable by entering it as a command. Use the variable in a few mathematical expressions just as you did in Task 7, and check its type.

You may notice that, even after using the variable in several expressions, its value does not change. For a numeric variable, we can change its value only through re-assignment, using the same notation as before. If it or other variables have already been defined then they can be used as part of the expression on the right-hand side; e.g., we can run `x = x + 1`, at which point (assuming `x = 5` was the most recent assignment) `x` has the value 6.

Task 16. Update the assignment of a variable, and create new variables that reference the value of other variables.

We have used simple variable names like `x` and `y` in the above examples, but in more complicated programs it is considered good practice to use meaningful names, like `num_respondents` (for a survey) or `avg_age`, to remind you of the purpose of these variables. A variable name must start with a letter, and can consist of letters (both upper- and lower-case), digits, and the underscore symbol `_`. There are a few reserved words in Python (e.g., `while` and `if`) that cannot be the names of variables, but otherwise anything goes.

8 Python Files

We're now ready to migrate from the interactive Python interpreter to creating standalone Python files, which have a `.py` extension. This is crucial for building up more complex programs, as well as for saving and sharing our code.

Task 17. Create a new file in your `lab01` directory called `test_file.py`, and add into it some of the numeric expressions you've written so far. Run this file by entering `python test_file.py` into the command line, or clicking the 'run' button in your IDE of choice.

Did nothing happen when you ran the program? So whereas when you type `4+5` into the interactive interpreter you see `9` now you get nothing? Don't worry, that's the expected outcome, and we'll figure out how to deal with this momentarily.

9 Basic Printing

Unlike with the interpreter, it is impossible to see what is happening within a file even when we run the code. To do this, we need to use a form of file *input/output* (often called i/o for short) to link the operations within the file with the outside world.

We will learn more advanced forms of file i/o later in the module, but for now we'll start with the most basic one, which is printing. In Python, basic printing is very simple: the word `print` is followed by a space, and then by whatever expression you want to print. (In Python3, the values you are printing now need to be inside parentheses.) The expression will then be evaluated when the command is run and its value will be printed. If you want to print the value of multiple expressions, you separate them with `,` . The values are then printed, one after the other, with a space between them.

Printable values have a new type, `str` (short for string), that we will learn about in much greater depth next week. For now, all you need to know is that a `str` starts and ends with either single quotes `''` or double quotes `""`, like `'hello'` or `"hello"`.

Task 18. In computer science, creating a program to say "hello, world!" is often considered the first step on the road to greatness. So, in keeping with the tradition, create a program `hello_world.py` that, when you run it, prints this string.

Python is somewhat unique in that `print` accepts arguments that have a type other than `str`. If you provide expressions with other types to `print`, like `int`, it first evaluates the expression, then casts the resulting value as a string before printing it.

Task 19. Try a few more advanced features of `print`, like giving it multiple arguments and/or complex expressions. Put these in a new file called `advanced_print.py`.

10 Comments

As you create more complex programs and share your code, it will become essential to add *comments* to your code to let both yourself and others know what it is doing. A comment in Python starts with `#` and continues until the end of the line.

Task 20. Go back to Task 19 and comment your code there to say what each `print` is doing by specifying the expected outcome.

11 Testing

While comments are incredibly important, the above usage of them is not really appropriate when we are trying to *test* the output of some expression; instead, comments are mostly used to explain the inner functioning of the program to someone looking at your code trying to figure out how it works (which is very often your future self who has forgotten how you wrote it!).

To instead convince someone that your program is functioning correctly (regardless of how it works), you should use `print` to do testing that is externally visible. Consider, for example, the following code:

```
print 'should return 9'
print 'is returning', 4 + 5
```

If someone runs this program, they can immediately look just at what is printed out to see if the expected result matches the actual result, and thus whether or not the code is behaving properly. This type of testing is especially important for *semantic* errors, in which the program runs but is silently producing unexpected behavior.

Task 21. Go back to Task 20 and, rather than specify the expected outcome in a comment, put it in a `print` statement. Run your code to ensure that all the tests pass.

Checkpoint 2. You have reached a checkpoint! Please call the PGTA or lecturer over to check what you've done so far.

12 Putting It All Together

Congratulations, you now know how to use Python as a calculator! This may not necessarily seem like much, and by the end of this module you'll of course be able to do far more, but for now let's celebrate this achievement.

There are many different things we can use calculators for, but let's play around with converting different scientific units, which is something that you may need to do frequently when comparing data across different datasets. For example:

- 1 mile = 1.61 kilometers
- 1 stone = 14 pounds = 6.35 kilograms
- 100 Celsius = 212 Fahrenheit (and in general $C = (F - 32) \cdot 5/9$)

Task 22. Create a file `units.py`. For each unit you consider (feel free to use ones other than the above examples), convert a variable in one scientific unit (e.g., `num_miles`) into its equivalent in another scientific unit (e.g., `num_kms`). Try to reuse code as much as possible; e.g., by recording the conversions themselves as variables, as in `secs_in_min = 60`. Explore several different types of unit conversions, and get into the habit of commenting and testing your code.

As a specific task, let's say I told you that I can run 800 meters in 2 minutes and 10 seconds

(I can't though!). What does that translate into in terms of my minute per mile pace? Bonus points if you express it in the usual way we express paces (e.g., 8:30 minutes per mile) as opposed to a decimal (e.g., 8.5 minutes per mile).

13 Handing In

Feel free to play around with Python some more, focusing in particular on any of the above aspects that you may have struggled with. Once you are done, you'll use the same procedure to hand in your work that we'll be using for the rest of the labs in this module.

First, make sure that all of the files you've created throughout this lab are in the `lab01` directory, and that this directory is contained inside your Git repository. To add these local files to the remote repository, navigate to the `lab01` directory and run `git add *.py` to add all of the files, or run `git add <filename>` for the following specific files:

- `test_file.py`
- `hello_world.py`
- `advanced_print.py`
- `units.py`

Add another file, `partner.txt`, to let us know, using their full name and/or student number, who your lab partner was this week. This needs to be done every week so that you both get credit.

Once these files are added to your local copy of the repository, commit these changes locally by running `git commit -m "<commit message>"`. Finally, push them to the remote repository using `git push origin master`. Feel free to revisit Section 3 for a reminder on how to do this, or call one of us over if you're really having trouble.

Once the code is checked into the repository, you're all done! As with all future labs, solutions should be on the repository for the module by the end of the day, both in terms of code files added to a `code` subdirectory of the directory for the lab (e.g., `lab01/code/`), and a `readme_solns.pdf` file for the lab with the solutions (or partial solutions, as they get longer) written inline. You'll also get feedback before next week, as a `mark.txt` file added to your personal repository, which you can see either by looking at the repository website on Github or by running `git pull origin master` to check for updates. You are encouraged to always look over the solutions, even if you were confident in your own, as there are usually many different ways to solve a problem and it is useful to see more of them.