Informatics 2 – Introduction to Algorithms and Data Structures

Lab Sheet 1: Getting started in Python

This is the first of three lab sheets which are intended to serve multiple purposes:

1. Introduce you to the basics of programming in Python (enough to enable you to tackle the coursework assignments).
2. Provide practical illustrations of some of the ideas introduced in lectures (many of the examples and exercises tie in with the lecture material).
3. Help you to think about what is going on 'under the hood' in the Python interpreter. E.g. what data structures are being used to implement lists or sets in Python, and what are their strengths and weaknesses? This is one aspect of what a good programmer should be aware of.

The lab sheets form an important part of the course, and you should take time to work through them even if you are already fluent in Python. This first sheet will help you to get started, and will introduce you to various kinds of (simple and complex) expressions.

The exercises appearing within the sheets are purely for your own private use: you are not required to submit them. Solutions to the exercises within each sheet will be issued a week after the sheet itself.

A more comprehensive tutorial for Python (version 3.11.5) can be found online at:

`https://docs.python.org/3/tutorial/index.html`

Feel free to check it if you feel that you need a more closely guided tutorial. The docs are also a very useful source for verifying or reminding yourself of how some things are done in Python, so do go back to them as often as you need.

1 Accessing Python

For the purpose of working through these lab sheets, you may either install Python on your own machine, or else work on an Informatics DICE machine.

1.1 Your own machine

If you don't have Python installed on your machine already, it's easy to do from the following website:

`https://www.python.org/downloads/`

We recommend that if possible you work with Python 3.11 or above (the latest version is 3.11.5).

Working on your own machine is fine for the purpose of these lab sheets. When it comes to the assessed courseworks, you may use your own machine to develop your code, but you should **check that it runs correctly under DICE** before you submit it, as that's where it will be tested and marked.

1.1.1 Editors

It is usually easier to write Python via some *editor*, that will have features like auto-complete, colour highlighting, embedded interpreter etc. There are many choices, and you are free to use whichever editor you would like. One easy-to-use and quite intuitive option is VS Code, which can be downloaded from the link below.

`https://code.visualstudio.com/`

Setting it up should be fairly straightforward, but feel free to ask for help in Piazza if it turns out to be more difficult than expected!

1.2 DICE machines

You will have in-person access to DICE machines in the Appleton Tower machine halls, but it's also possible to log on to them remotely. You may do this via an ssh gateway, or using the graphical Remote Desktop Service (XRDP). Information on connecting by these two means may be found on the School's Computing Support pages:

```
https://computing.help.inf.ed.ac.uk/external-login
https://computing.help.inf.ed.ac.uk/remote-desktop
```

Once you've connected, open a command shell on the DICE machine, and type

```
python
```

This launches the Python interpreter on DICE.

2 Interpreter

In this first lab sheet, we'll be executing commands directly in the Python interpreter. The ability to do this is one of the nice things about Python. This can save time when you are developing software: when you are not sure about something, you can often test it directly from the interpreter, without the overhead of having to write and compile a small program. (In the second lab sheet, we'll consider programs written and maintained in a separate source file.)

⋆ In this lab sheet, type the lines introduced by `>>>` and by `...` You may cut and paste individual lines if you prefer, but cutting and pasting whole sections will usually not work (we'll see why).

3 Basic Types

3.1 Numbers

The Python interpreter can be used as a simple calculator. Expression syntax is straightforward: the operators `+`, `-`, `*`, `/` work just like in most other languages.

⋆ Enter the following expressions in the Python interpreter you have just launched:

```
>>> 2+2
>>> 3*2
>>> 2 + 7.3
```

The `_` variable can be used whenever you wish to utilize the value produced by the latest calculation in the interactive interpreter:

```
>>> 3*5
>>> 1 + _
>>> _ / 4
```

Lines previously inserted in the interpreter shell can be recalled by pressing the ↑ cursor button. The ← and → cursor buttons also work as expected to move backwards and forwards through the characters in the current line. Take a moment now to experiment with these keys. You can type `exit()` to exit the interpreter.

`%` normally stands for the *modulo* operator, which calculates the remainder of the integer division between its two operators. This operator is more useful and more common than you might think.

```
>>> 153 % 10
>>> 37 % 6
>>> 231 % 3
```

Similarly, the `//` operator performs integer division, returning only the integer part of the quotient:

```
>>> 37 // 6
>>> # If you change 37 and 6 for any other two numbers,
... # this will always return the first number.
... 6 * (37 // 6) + (37 % 6)
```

Finally, the `**` operator performs exponentiation:

```
>>> 2 ** 2
>>> 2 ** 3
>>> 10 ** 5
>>> 1.5 ** 4.3
```

Note that, unlike other more low-level languages like C and Java, integers in Python can be arbitrarily long while maintaining full precision:

```
>>> 123456789123456789123456789123456789123456789123456789 * 987654321
```

This is interesting from a data structures point of view, and may present some interesting questions about the implementation if you think about it, but we will not dig deeper for now.
Floating point numbers (scientific notation), however, have limited precision. The limitations in precision may appear in situations you do not expect it to. For example, try this:

```
>>> 1.2 - 1.0
>>> _ - 0.2
```

Note that exact division (`/`) will produce floating point numbers even when dividing two integers, even when they are exactly divisible. This may reduce precision inadvertently:

```
>>> x = 123456789123456789123456789123456789123456789123456789123456789
>>> x
>>> y = x * 987654321
>>> y
>>> z = y / 987654321
>>> z
>>> x == z
```

As a consequence, you should be careful whenever you use floating point numbers or `/`, and use alternatives whenever precision is important. This happens in any programming language that uses floating point numbers[1].

3.2 Boolean Types

The Boolean type is named bool. Its values are True and False[2]. The `bool` function takes any Python value and converts it to True or False, in the way that you would expected in, for example, C.

```
>>> bool(1)
>>> bool(0)
>>> bool([1])
>>> bool([])
```

[1]If you want to know why, read up on the implementation of floating point numbers.
[2]Capitalization is important!

3.3 Strings

Strings can be enclosed in single or double quotes[3]:

```
>>> 'hello'
>>> 'how\'s it going?'
>>> "how's it going?"
>>> 'This is "strange"'
```

Strings can be concatenated using the `+` operator, and can be repeated with the `*` operator:

```
>>> 'Hello ' + ', ' + 'how are you'
>>> 'help' + '!'*5
```

The individual characters of a string can be accessed using indices. For example the first character has index 0. Substrings can be specified with *slice* notation, two indices separated by colon. When using slices, indices can be thought of as pointing **between** characters, instead of pointing to characters: the left edge of the first character is 0 and so on. Another way to think about this is that the initial index is inclusive and the final index is exclusive.

The following commands illustrate this with the help of a user-defined variable called `word`. Take care when you name your variables not to use reserved words of the language, such as `if`. For a list of reserved words, look here: `https://docs.python.org/3/reference/lexical_analysis.html#keywords`.

```
>>> word = "hello"
>>> word[0]
>>> word[2]
>>> word[0:2]
>>> word[2:5]
>>> word[:2]
>>> word[2:]
>>> word[:]
```

Negative indices start counting from the right end.

```
>>> word[-1]
>>> word[-2:]
```

Strings are *immutable.* This means that they cannot be modified once they are created. Trying to modify a substring results in an error. Numbers are also immutable, but this is true in (almost) every language. However, it is easy to create a new string by concatenating substrings from other strings, the same way that you create a new number by multiplying other numbers. String variable values can be replaced with new strings, and you should make sure you understand the difference between this and modifying the string itself:

```
>>> x = "hello"
>>> y = x
```

[3]There is no difference between them except when the string itself contains quote characters. Whichever quote character is being used to demarcate the string will need to be *escaped*, i.e., preceded with a backslash.

The following creates a new string and replacing the value of the variable `x` for it.

```
>>> x = x + ", how are you?"
>>> x
>>> y
```

Even the following is creating a new string. `x +=` is just 'syntactic sugar' for `x = x +`. They are no different.

```
>>> x += " I'm fine, thanks!"
```

However, the following is an attempt to modify part of an existing string, and results in an error. This is what is meant by saying *strings are immutable.*

```
>>> x[0] = "H"
```

The built-in function `len` gives the length of a string:

```
>>> len(x)
```

3.4 Lists

There are different types of compound structures in Python. The most versatile is the list.

```
>>> L = ['monday','tuesday','wednesday','thursday','friday']
>>> L
```

Note that, in Python (unlike Haskell), lists can contain elements of different types, and these elements may themselves be lists:

```
>>> [1,"abc",[2,[[]]]]
```

Like strings, list items can also be accessed using indices. Similarly, they can be sliced and concatenated:

```
>>> L[0]
>>> L[3]
>>> L[1:]
>>> L[:2]
>>> L[1:3]
>>> L + ['saturday','sunday']
```

This is because both strings and lists (and tuples and a few others we will not mention here) are what Python calls *sequence* types. Operations like verifying membership, concatenation, indexing, slicing and measuring the length of a sequence can be performed on any sequence type. For example:

```
>>> len(L)
```

One can verify the membership of an element to a list using the keyword `in`.

```
>>> 'wednesday' in L
>>> 'sunday' in L
```

Unlike strings, lists are *mutable*. Items can be added at the end of a list using the `append(item)` method of the `list` object.

```
>>> L3 = []
>>> L32 = L3
>>> L3s = [L3,L3,L3]
>>> L3.append(1)
>>> L3.append(2)
>>> L3
>>> L32
>>> L3s
```

What do you notice by running the code above? Perhaps we meant to add elements to L3, but those were also added to L32, and each “occurence” of L3 in L3s. This has to do with the way Python is referencing those lists, when using the ”=” operator. To avoid such effects, explicit copying (or *cloning*) of lists is sometimes important (but it makes the computer do work, so it should be done sparingly). A common way to do this in Python is by using the slice operator (`:`) already introduced. For example:

```
>>> L0 = []
>>> L1 = L0[:]
>>> L1.append(1)
>>> L2 = L1[:]
>>> L2.append(2)
>>> L0, L1, L2
```

`insert(i,x)` inserts the item x at position i, moving every element after that to the right.

```
>>> L2 = ['a','b','d','e']
>>> L2.insert(2,'c')
>>> L2
```

`remove(x)` removes the first item in the list whose value is x, moving every element to its right to the left.

```
>>> L2.remove('d')
>>> L2
```

`pop()` returns (and removes) the last item in the list; likewise, `pop(i)` returns and removes the item in position `i`.

```
>>> L2.pop()
>>> L2
```

`reverse()` reverses the elements of the list, in place.

```
>>> L2.reverse()
>>> L2
```

You may iterate over a list directly:

```
>>> for v in ['a','b','c','d']:
...     print(v)
...
```

⋆ This is our first example of a Python command spread over more than one line of input. The '`...`' prompt indicates that Python is expecting more. In this instance, supplying a blank line at the end (i.e. just hitting return again) tells the interpreter that you've finished. A reminder that you should either type the input yourself, or cut and paste one line at a time. (If you try to cut and paste all the above at once, you'll see what goes wrong.)

Note that in Python, **formatting has meaning** - this means the spacing is very important. Groups of statements are often indented under a header: for instance, the `print` statement in the second line above is indented relative to the `for` statement. Blocks of code are nested by increasing the indentation. There are no end-of-statement symbols (like the semicolon; in Java or C). Instead, newline marks the end of a statement.

If you want to retrieve the index and the value at the same time, use the `enumerate(list)` function.

```
>>> for i, v in enumerate(['a','b','c','d']):
...     print(i, v)
...
```

Again, the additional (unindented) blank line indicates the end of the `for` loop.

Lists in Python are one of the most fundamental tools that you will be working with. So much so, that a fundamental functionality of Python is what is called the *list comprehension syntax* (this is similar to what you may have seen in Haskell). This allows us to define lists through an expression over other lists:

```
>>> [2*x for x in range(1,6)]
>>> [y*y for y in [2*x for x in range(1,6)]]
>>> [(x,y) for x in range(2,8) for y in range(5,9)]
>>> [(x,y) for x in range(2,8) for y in range(5,9) if x < y]
```

Notice that the last example shows how a list can be 'filtered' so as to contain only those members satisfying the `if` condition.

Of course, all this is mostly syntactic sugar combining for loops with lists, as any of the above could be expressed using imperative style for loops:

```
>>> L1 = []
>>> for x in range(1,6):
...     L1.append(2*x)
...
>>> L1
```

So, what are the advantages of using list comprehension? The "spelled out" version with the for loop above is more verbose, and requires us to define extra helper variables, distracting us from a very simple notation: wanting the double of each element in the list. But please tread carefully! You can condense for loops into rather complex-looking single-line statements. This is absolutely fine as long as you understand them. Single-line statements are much harder to debug, and if you simply copy-paste them from the internet, they might not do exactly what you need them to. If you are not sure, start with the loop-version and work it into the list comprehension version. Once you have a good command of list comprehension, you are very much encouraged to use it. The exercises in Section 3.6 below are a very good starting point.

3.5 Tuples

A *tuple* is composed by a number of values separated by commas, and enclosed by parentheses.

```
>>> T = (1,2,'three')
>>> T
>>> T[2]
```

Tuples can be nested.

```
>>> T1 = (1,2,(3,5))
>>> T1
```

What's the difference between tuples and lists? Tuples, like strings, are immutable and cannot be changed once created. If you try, you will get an error message.

```
>>> s = "hello"
>>> s[1] = "u"
>>> T1[2] = 3
```

But this does not mean that tuples have no reason to be used: sometimes you want to make sure that the tuple will not change and, perhaps more importantly, there are some important data structures that rely on the fact that items are known to be immutable (see below). The key is to know what you need in each occassion and choose the appropriate type.

3.6 Exercises on list comprehension

List comprehension is a very powerful tool, and it is great for expressing some of the most fundamental notions underlying an algorithm. Try these exercises to get some practice in it.

Exercise 1:
Define a list containing all even numbers between 5 and 134.

Hint: You can check if a number `x` is even by checking that `x % 2 == 0`.

Exercise 2:
Define a list containing all initial substrings of the string “python” (e.g. “py”, “pytho”, etc.).
Define another list containing all of its final substrings (e.g. “thon”, “n”, etc.).
Define another list containing all of its substrings (e.g. “py”, “thon”, “ytho”, etc.).

Exercise 3:
Create a list consisting of all triples `(x,y,z)` of three positive integers that add up to exactly 10.

Exercise 4:
To produce multiples of a number (e.g. 5), we can proceed in at least two very different ways:

- We can produce numbers and check if they are multiples of 5 (`x % 5 == 0`).
- We can produce numbers and multiply them by 5, which guarantees that the result is a multiple of 5.

Which of these options seems better to you? Write list comprehension definitions for the list of all multiples of 5 between 1 and 1000 calculated in each of these two ways.

Exercise 5:

Finding divisors of a number is harder than finding multiples. How would you define a list containing all divisors of 714 which are not 1 and 714?

Exercise 6:

Remember that a prime number is a number that is only divisible by 1 and itself. Define a list containing all prime numbers between 2 and 1000.

Hint: Use the previous exercise, and don’t worry here if your solution seems needlessly inefficient. To check if a list is empty, you can either do `len(list) == 0` or `list == []`.

3.7 Lists and complexity

Lists are a very important data structure in most programming languages, but there are many different ways to implement them, with different complexity properties for the different operations we may need to do with them. We will now do a few experiments to see this.

In Python, lists are implemented using *extensible arrays*. Simplified, this means that Python pre-allocates more space in memory than is strictly needed to hold a list, so that new insertions can be performed without having to re-allocate memory, and keeping the entire list sequentially in a contiguous space in memory. This makes accessing the list and modifying its elements quicker. When the pre-allocated space runs out, Python allocates a new, larger block (9/8 times bigger, roughly speaking), moving the entire list to the beginning of the new block and freeing the old block. Similarly, when elements in the list are removed, sometimes the list will be moved to a smaller space to free the unused space. Usually all happens without you noticing, but we can do some experiments to try to see it in action.

First, let's set up a tool to measure running time of computations. Type the following at your terminal (and don't worry too much about the details of this):

```
>>> import timeit
>>> def time(com):
...     return timeit.timeit(com, number = 1, globals = globals())
...
```

This defines a `time(com)` function that allows us to measure the time a computation takes to happen (in seconds). This time depends, of course, on many things, such as the speed of the computer we are running on, how many things are running at the same time, etc., but it gives us a rough idea of how long things take to run. Try it. Note that measuring the same computation multiple times may produce different results:

```
>>> time('[1,2,3][0]')
>>> time('[1,2,3][0]')
>>> time('[1,2,3][0]')
>>> time('[1,2,3][0]')
>>> time('[1,2,3][0]')
```

So repeating the measurements is encouraged, to get a better idea of the average time it takes to do things.

As we said, accessing and modifying elements in a list in Python is more or less independent of the size of the list and how far in the list we look for elements, so the following should all take approximately the same amount of time:

```
>>> L1 = list(range(1,100))
>>> L2 = list(range(1,10000000))
>>> time('L1[0]')
>>> time('L1[50]')
>>> time('L1[98]')
>>> time('L2[0]')
>>> time('L2[98]')
>>> time('L2[1337]')
>>> time('L2[5000000]')
>>> time('L2[9999998]')
```

And similarly for the following:

```
>>> time('L1[0] = 5')
>>> time('L1[50] = 3')
>>> time('L1[98] = 777')
>>> time('L2[0] = 1234567')
>>> time('L2[98] = 7654321')
>>> time('L2[1337] = 0')
>>> time('L2[5000000] = -1')
>>> time('L2[9999998] = -500')
```

Appending an element to the list is almost always fast:

```
>>> L1 = list(range(1,100000))
>>> time('L1.append(0)')
>>> time('L1.append(0)')
>>> time('L1.append(0)')
>>> time('L1.append(0)')
>>> time('L1.append(0)')
>>> time('L1.append(0)')
>>> time('L1.append(0)')
```

However, sometimes it will trigger the re-allocation of the list, taking considerably longer. While it is hard to see this happening clearly, we can do some things that show us that something is happening underneath. Measure the time it takes to append an element 300000 times, and then look for the maximum and minimum values in these times. Notice how they are (on my computer) nearly 4 orders of magnitude different:

```
>>> times = [time('L1.append(0)') for n in range(1,300000)]
>>> min(times)
>>> max(times)
```

It is a lot easier to see how time for insertion is clearly slower for inserting into earlier positions of the list:

```
>>> L2 = list(range(1,10000000))
>>> time('L2.insert(2,0)')
>>> time('L2.insert(9999999,0)')
```

This is, of course, because Python needs to move all the elements after the position being inserted one space forward. The same happens for removing elements:

```
>>> time('L2.pop(2)')
>>> time('L2.pop(9999900)')
```

3.8 Sets

Lists may have repeated elements, and the order within them matters. So that the list `[1,2,1]` is different from `[1,1,2]` and both are different from `[1,2]`. Often, we wish that all these were considered equal: we only want to know whether an element is or is not in the list, irrespective of how many times or in what order. This is where *sets* come in.

Sets are a *mutable* data structure that corresponds essentially to the mathematical concept of a (finite) set. To define a set in Python we do:

```
>>> s1 = {1,2,3}
>>> s2 = {1,2,3,1}
>>> s1
>>> s2
>>> s1 == s2
```

To produce an empty set, we use `set()`:

```
>>> set()
```

Sets are sequences, and so all functions that work in sequences work in them, like `len(set)` or the `in` keyword.

The `add(element)` method can be used to add elements to a set. Note that adding an element already in a set leaves it unchanged.

```
>>> s = {1,2,3}
>>> s
>>> s.add(4)
>>> s
>>> s.add(4)
>>> s
```

Similarly, you can remove elements from a set using the `remove(element)` or the `discard(element)` methods.

```
>>> s = {1,2,3}
>>> s.remove(2)
>>> s
>>> s.discard(2)
>>> s
```

You can extract one (unpredictable)[4] element from a set by using the `pop()` method:

```
>>> {3,5,1,2}.pop()
```

Sets have several useful binary operations you can perform on them (union, intersection, difference and symmetric difference, to name a few). We will not go in detail through these but you are welcome to look them up, and perhaps decide to use them in some of your programs. Note that all of these operations create new sets.
You can also check if a set is contained in another by using the `<=` operator:

```
>>> x <= y
>>> y <= x
>>> x <= (x | y)
>>> (x & y) <= y
>>> set() <= x
```

Like lists, sets can contain heterogeneous elements:

[4]It is not really random, but it is incorrect to presume you will know which one it will be.

```
>>> {1,2,'a',84.37}
```

However, an important difference between sets and lists is that sets may only contain *immutable* elements. (The same also applies to dictionaries: see below.)

```
>>> {[1,2]}
>>> {{1,2}}
```

Note that a set *itself* is a mutable structure – elements can be added or removed – but the individual elements themselves must be immutable. This is one place where *tuples* come into their own: unlike lists, they can be included as members of sets.

```
>>> {[1,2]}
>>> {(1,2)}
```

Sets, like lists, can be defined using comprehension syntax:

```
>>> {2*x for x in {1,2,3,4}}
>>> {x % 2 for x in {1,2,3,4,5,6,7,8}}
```

In fact, comprehension for lists and for sets work well together. For instance, it is simple to define a set from a list:

```
>>> L = [1,2,3,2,1]
>>> [2*x for x in L]
>>> {2*x for x in L}
```

Lists can also be defined from sets, but this has a big caveat: in which order? It will often not be the one you will expect:

```
>>> L1 = [x for x in {1,2,3}]
>>> L2 = [x for x in {3,2,1}]
>>> L1
>>> L2
>>> L1 == L2
```

In fact, the same set can be printed in different ways depending on how you defined it!:

```
>>> s1 = {1,2,1.5}
>>> s2 = {2,1.5,1}
>>> s1
>>> s2
>>> s1 == s2
```

which can cause mayhem when creating lists from them:

```
>>> L1 = [x for x in s1]
>>> L2 = [x for x in s2]
>>> L1
>>> L2
>>> s1 == s2
>>> L1 == L2
```

All of these facts trace down to implementation notions that have to do with performance in the way these data structures are used and that we will talk about in this course, and it may even change between implementations of the Python interpreter, different computers or even different moments in which you execute it. The general principle is: whenever you use a set (or a dictionary), make sure that your program's correctness is independent of the order in which elements of the set are considered by the program.

3.9 Dictionaries

Dictionaries are very useful structures; in some other languages they are referred to as *maps*. They are indexed by **keys**, unlike lists which are indexed by numerical indices. Keys can be any immutable objects, but not mutable ones (we'll see why in a moment).

A dictionary can be seen as a set of *key:value* pairs, with the constraint that keys need to be distinct. The main operations performed on dictionaries are storing and retrieving values by their keys.

```
>>> num = {'one':1, 'two':2, 'three':3, 'four':4}
>>> num['three']
>>> num
```

You can easily add a new item or modify an existing one in a dictionary:

```
>>> num['five'] = 6
>>> num
>>> num['five'] = 5
>>> num
```

You can delete an item from a dictionary using the built in function `del(item)`:

```
>>> del(num['three'])
>>> num
```

To list all the keys from a dictionary, you use the method `keys()`:

```
>>> num.keys()
>>> list(num.keys())
```

The valued returned by `keys()` is a generator, which we will not cover here. To transform it into a list, simply surround it with the `list(generator)` function.

If you iterate directly over a dictionary, it iterates over its keys:

```
>>> for k in num:
...     print(k)
...
```

You can iterate over a dictionary, retrieving keys and values at the same time using the `items()` method:

```
>>> for k, v in num.items():
...     print k,v
...
```

Both sets and dictionaries in Python are implemented using a data structure called a *hash table*, which we will cover in lectures. Hash tables are very efficient, but they only work properly when the elements in question are immutable: this is the main reason why only immutable elements are permitted in sets or as dictionary keys.

⋆ You may now end your interpreter session by typing `exit()`. Then close the connection by typing `logout` in the shell, or by using the keys `Ctrl+D`, and enjoy the rest of your day!