Informatics 2 – Introduction to Algorithms and Data Structures

Lab Sheet 2: Writing programs in Python

In this lab sheet, we move on from writing simple one- or two-line commands to writing some actual programs in Python. Just as in Haskell, programs frequently take the form of *function definitions*, so this is where we'll start. We'll make use of the material covered in Sheet 1, so you may find it useful to refer back occasionally.

1 Writing, saving, loading and running programs

For the previous Lab sheet, we have used the interpreter directly in the shell. In this Lab, we will create and manipulate a Python *script*. If you are working remotely on a DICE machine, open a shell and do the following:

⋆ In the shell, create a new directory called "`MyPython`":

```
mkdir MyPython
```

An empty directory has been created and you can see it using the `ls` command in the shell which displays all content of the current location.

⋆ Go to this directory:

```
cd MyPython
```

At this point you might want to write the commands as a program in an editor (such as nano, emacs, vim, gedit, or VS Code, which we suggested in Lab Sheet 1), so that you can easily modify the lines in case of errors. You can then execute the programs from the shell or import them from the interactive interpreter.

⋆ For example, to open the *nano* editor, type the following command from the shell:

```
nano my_program.py
```

The editor shows the content of your file, which is now empty but we will add few lines of code. For now, let's start with a small function example. The following function returns the square value of a given input value.

⋆ Try typing the following code into your file editor:

```
def square(value):
   """Returns the square of the value"""
   return value*value
```

Save the file using `Ctrl + S`. You should be able to see your file in the current folder using the `ls` command.

It is good practice to use a text editor for editing your code as you test it. One easy way to get this done *remotely* is to open two terminal shells, one for launching the interpreter, and one for the text editor. In this way, you won't need to close the text editor each time you launch the interpreter, you can simply save the code and reload the program you have just edited in the interpreter. If you are running Python on your own machine, then you can work with the local text editor and your terminal side by side, or. Your editor might also have an integrated terminal.

⋆ Now (in a different terminal shell, where you have sshed beforehand) we are going to launch the Python interpreter and import the python file.

```
python
```

Note: If you are on a Mac, you might need to use "`python3`" rather than "`python`".

```
>>> import my_program
```

This loads the contents of `my_program.py` into the current session. Notice, however, that we do not include the `.py` file extension when using `import`.

When we import a program module by this method, by default we will have to qualify all elements defined in the module with the module name to access them. For example:

```
>>> my_program.square(5)
```

However, we can also import specific elements into the main scope like this:

```
>>> from my_program import square
>>> square(5)
```

If you decide to change your program source file using the editor running in parallel, you may want to run the updated version of your code without leaving the Python interpreter. There is a `reload()` function for this in Python, which reloads an imported module. To run the updated version of your code in the interpreter, type the following:

```
>>> import importlib
>>> importlib.reload(my_program)
```

Another way of 'loading' a program into the interpreter is simply to type in the interactive shell over several lines:

```
>>> def square(value):
...     """Returns the square of the value"""
...     return value*value
...
>>> square(23)
```

However, this method shouldn't be used for programs of any length, as it doesn't make it easy to modify your program and try again.

2 Function definitions

Now let's look a bit more closely at the structure of the above definition of the `square` function.

A function is introduced by the keyword `def`, followed by the function name and by its list of parameters in parentheses. The statements that form the function body start on the next line, and are indented. Remember: in Python, formatting has meaning – this means the spacing is very important. Groups of statements are indented under a header. 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.

The `return` keyword is used to return a value as a result of the function.

The first line of a function definition can optionally be a string that describes the function. This string can be used by automatic generators of documentation.

To check your understanding of the function definitions, using the text editor, create a new function `cube` similar to the square, and import it in the interpreter after reloading the program. Call the `cube` function on few integers and see if you get the results you expected.

3 Flow control statements

Now we will introduce the `if`, `while` and `for` keywords for flow control.

3.1 If statement

⋆ Using an editor, create a file `binary.py` containing the following lines:

```
x=input("Enter a binary sequence : ")
if (x[0] == '0'):
    print("Starts with 0")
elif (x[0] == '1'):
    print("Starts with 1")
else:
    print("Error: Not a binary number!")
```

⋆ After saving the file, launch it from the shell.

```
python binary.py
```

⋆ When asked, insert a binary sequence (such as 00101 or 10010) and press `Enter`.
It is possible to create more complex conditions using the keyword `and` for conjunction and the keyword `or` for disjunction – e.g.

```
if (x[0] == '0' or x[0] == '1'):
    print("Starts with " + x[0])
else:
    print("Error: Not a binary number!")
```

3.2 While loop

⋆ Using the editor, create a file `downtoone.py`. Type the following lines.

```
x = 10
while (x > 0):
    print(x)
    x = x - 1
```

⋆ Save the file and then launch the program from the shell.

```
python doontoone.py
```

3.3 For Loop

The `for` statement iterates over the items of any sequence (such as strings, lists or sets), using the keyword `in` discussed earlier.

⋆ Using an editor, create a file `printlist.py` . Type the following lines.

```
L = ['monday','tuesday','wednesday','thursday','friday']
for x in L:
    print x
```

⋆ Save the file and then launch the program from the shell.

To iterate over a sequence of numbers, the built-in function `range()` can be used to automatically generate a sequence of integers. Strictly speaking, `range` produces an object known as a *generator*. We won't go into detail here on what generators are: all that matters here is that a generator can be transformed into a list using `list`, and can be iterated over using `in`. The next few examples illustrate these features.

⋆ Re-enter the python interpreter and type in the following lines.

```
>>> range(5)
>>> list(range(5))
>>> range(5,10)
>>> list(range(5,10))
```

⋆ Using an editor, create a file `printnumbers.py`. Type the following lines.

```
for x in range(10):
    print(x)
```

⋆ Save the file and then launch the program from the shell.

You can write an `if` statement nested inside a `for` statement as follows, which prints the even numbers less than 10. Remember that nesting is indicated by increased indentation.

⋆ Using an editor, create a file `printevennumbers.py` and enter the following lines:

```
for x in range(10):
    if (x%2 == 0):
        print x
```

⋆ Save the file and then launch the program from the shell.

4 More about functions

4.1 Recursion

While technically not a flow control statement, function recursion serves a very similar purpose to `while` loops. Remember that a function is called *recursive* if it calls itself.
The typical example is the factorial function. The factorial $n!$ of a number n is defined as:

$$n! = n \times (n-1) \times (n-2) \times ... \times 2 \times 1$$

so that, for example, $3! = 6$ or $5! = 120$.

The factorial can be defined with `while` or `for` loops[1], but it is often easier to define with recursion. Notice that for any $n \geq 2$, it is true that $n! = n \times (n-1)!$. The factorial of 1 is simply 1. Actually, it is usual to extend the definition to $n = 0$ so that $0! = 1$. We can use this to implement it recursively. First, we need to write down the *base case*: the case in which the recursion ends. Every recursive function needs a (or multiple) base case(s), otherwise it would never terminate! The base case in this case is `n == 0`:

```
def factorial(n):
   if n == 0:
      return 1
```

and then add the recursive case: multiply n by the factorial of $n-1$, calling recursively:

```
def factorial(n):
   if n == 0:
```

[1] In fact, technically every recursive function can be implemented using loops, but **do not do this**. Use recursion when it makes sense.

```
    return 1
  else:
    return n * factorial(n-1)
```

Go back to the interpreter and check that it works:

```
>>> factorial(3)
>>> factorial(5)
>>> factorial(100)
```

Now a little exercise in using recursion.

Exercise 1:

Recall Euclid's algorithm for computing the *greatest common divisor* of two integers, as illustrated in the Lecture 1 slides. This relies on two mathematical facts:

$$\mathrm{GCD}(n, 0) \;=\; n\,, \qquad \mathrm{GCD}(m, n) \;=\; \mathrm{GCD}(n,\, m \bmod n)$$

Use these facts to write a recursive function `GCD(m,n)` that computes the GCD of any given integers $m \geq n \geq 0$. Try out your function on some large numbers.[2]

Recursion will play an important role in this course. However, it's worth noting that recursion is not so ubiquitous in Python programming as it is in Haskell. The Python interpreter is not highly optimized to deal with large numbers of recursive calls. The default limit on the nesting depth of function calls is around 1000. Should you wish to increase this limit, this can be done as follows (though your program may crash if you set the limit higher than your installation can support!)

```
>>> import sys
>>> sys.setrecursionlimit(5000)
```

4.2 Higher-order functions

In Python, functions are *first-class elements*. This means that they can be treated as values, stored into variables, and passed as arguments to other functions. Functions that receive other functions as arguments are often called *higher-order functions*.

Let's see a simple example. Define the function `twice` that applies a function two times to a value (applies it to the value, and then applies it again to the result):

```
def twice(f,x):
  return f(f(x))
```

[2] *Surprising fact of the day.* What is the probability that two whole numbers chosen at random will have 1 as their GCD (i.e. will have no factors in common)? The answer turns out to be $6/\pi^2$: that is, about 0.607927. [More rigorously, $6/\pi^2$ is the *limit* as n tends to infinity of the probability that two numbers chosen uniformly and independently at random from $\{1, \ldots, n\}$ will have no factor in common.] Who would have guessed that π had anything to do with it?

Now, from the interpreter, try running this function on `factorial` and `3`.

```
>>> twice(factorial,3)
```

You should obtain as result `720`. Why? Well, `twice(factorial,3)` is the factorial of the factorial of 3. The factorial of 3 is 6, whose factorial is 720, the result. But we can use the same `twice` function to double any other function. For example, try using the `exp` function in the `math` library, which calculates e^n for the given n:

```
>>> from math import exp
>>> twice(exp,3)
```

This will yield the value $528491311.4854943 = e^{(e^3)}$.

Being able to pass functions as arguments is, however, very useful for this course. Often, an algorithm is specified partially, with some parts of it being unspecified, allowing the user of the algorithm to provide their own implementation of them. For example, in search and sort algorithms the comparison function is often unspecified. The function that implements the search can take a comparison function as one of its arguments, and apply this to the list elements it finds (see Exercise 2(4) below).

5 Further exercises

The following programming exercises will give you some practice both with Python itself and with aspects of the lecture material. They will help you to ensure that you are at an adequate level of understanding for the current stage of the course.

Exercise 2:

In this exercise, we will focus on *modular exponentiation* as covered in Lecture 2. Modular exponentiation consists of calculating powers of integer numbers modulo another integer number. In part 5 of this exercise we will get a glimpse of why this is important. Let's first begin with some more simple functions.

1. Write a function `expmod1` which accepts integers a,n,m and returns the value of a^n mod m, computed using the built-in operations for exponentiation (`**`) and modulo (`%`).

   Some examples to check that your function is correct:

   ```
   >>> # Should return 1
   ... expmod1(2,8,3)
   >>> # Should return 36
   ... expmod1(135,202,53)
   ```

2. Write a function `expmod2` that computes the same operation with a different method. Start with the number 1, and multiply this by a, n times, reducing modulo m each time you multiply. (Use a `for` loop). The results returned by this function should be the same as for `expmod1` (for modest values of a, n, m).

3. Write a third function `expmod3` that computes the same operation again using the "fast" method mentioned in Lecture 2. (Use a recursive function). Try the same examples again.

4. Do some timing experiments to compare the feasibility and efficiency of these three implementations on numbers a, n, m of various sizes. Can you explain why the differences appear? Besides the time taken to compute the result, what other issues may be of concern?

5. The fast method for modular exponentiation is of considerable practical importance, for instance in the implementation of the *RSA cryptosystem* which is discussed in the DMMR course. Here we'll briefly consider how it can be used for almost-guaranteed *primality testing*, which is itself an important ingredient of RSA.

   Fermat's little theorem tells us that if n is prime then $2^{n-1} \bmod n = 1$. So if `expmod3(2,n-1,n)` returns anything other than 1 then we know that n is definitely composite (even though we may have no idea what its factors are!). Conversely, if `expmod3(2,n-1,n)` does return 1, and n is large, it is extremely probable that n is prime, though not absolutely guaranteed (there are a sprinkling of exceptions).

   Use these ideas to write a function `probableprime(a)` that can quickly locate a probable prime number greater than or equal to a given starting value. This function should work quickly even for very large numbers (100+ digits).

   Here are a few example results so that you can verify your function. All of them should run within a few seconds at the very most.

```
>>> # Should return 5
... probableprime(4)
>>> # Should return 17
... probableprime(14)
>>> # Should return 149
... probableprime(143)
>>> # Should return 454056225438947917
... probableprime(333**7)
>>> # Should return
... # 1299119025548714519410320843962351377546578201012739238437
... # 9012704624259433055094648925678485362472902010613951564738
... # 4910944921186523865849056275359066262352911682504769929309
... probableprime(1234**56)
```

Note: Fermat's little theorem, RSA and the other topics mentioned here are not strictly part of the syllabus of this course, but are very good real world examples of the importance of algorithm design.

Exercise 3:

In this exercise, we will talk about sorting algorithms.

1. Look at the pseudocode for in-place *insert sort* given in the Lecture 2 slides, and translate this into a Python function definition ($\leq$ 12 lines of code). You may use the built-in Python comparison operation `<`. (Note that `<` can be applied to both numbers or strings, so the same program will be able to sort a list of numbers into numerical order, or a list of strings into dictionary order.) But **don't** use Python's built-in sorting method `sort`, that would be cheating.

2. Now do the same for the pseudocode for *merge sort* ($\leq$ 25 lines of code, including a helper function for merging two already sorted lists into a new sorted list). A simple implementation will suffice: don't worry about trying to optimize for memory consumption. Use recursion.

3. Use `timeit` to compare the time efficiency of the two sort functions on lists of various kinds and various lengths.

   Here's how you can generate random lists of integers for testing purposes:

   ```
   from random import randint

   [randint(0,10000) for i in range(100)]
   ```

   But you should also test your functions on lists of other kinds: e.g. on lists that are already sorted.

4. Modify both your sorting programs to create *higher-order* functions which, instead of using the built-in comparison `<`, use a two-argument comparison function `before` which is supplied by the user as an extra argument to the sorting function.

   Construct an example to show how these higher-order versions can be used.