KIT103/KMA155 Practical 01: Introducing Python and its structured types

This lab is an opportunity to familiarise yourself with the interactive Python environment and to practise Python's syntax for defining structured types like sets. The main tasks are indicated by task. It will probably take longer than the time available in the lab, but everything here is entirely safe to do at home.

This and subsequent labs include embedded solutions that you can reveal when you want to check your own attempt or have been trying for a while and cannot work out the answer.

1 Spyder

Start Spyder, the Scientific PYthon Development EnviRonment (yes, they tried a bit too hard to get the name). Spyder is one of a number of alternative development environments for Python and has many good features. Once it starts you'll be presented with a code editing window on the left and, at the bottom right, an interactive environment, which is a good place for quickly testing new ideas. The interactive environment that Spyder provides is called IPython, and is one of many alternative interactive environments for Python. It's in this interactive environment that you'll do much of your work in this unit. To create more permanent scripts put your code into the editor on the left side of the screen and save the file.

Task: Have a quick play before moving on to the next section. Try typing some mathematical expressions, like `13 / 4`, `13 // 4` (yes, this does something different) and `13 % 4` (13 modulo 4), then try assigning those to variables, as in:

```
d = 13 / 4
q = 13 // 4
r = 13 % 4
```

and then enter those variable names at the prompt (`In [n]:`) to see what value they hold.

2 Tuple, list and set literals

A *literal* is any piece of program text that literally represents a value. Refresh your knowledge of the syntax for defining tuples, lists and sets based on these examples containing the values 1, 2, 3 and 1 (again). Note that # starts a single-line comment that has no effect on the computation.

```
t = (1, 2, 3, 1)     #t is a tuple of four elements
u = 1, 2, 3, 1       #u is also a tuple (and is equal to t)
l = [1, 2, 3, 1]     #l is a list of four elements; l != t but l == list(t)
s = {1, 2, 3, 1}     #s is a set of three elements
```

The value assigned to `s` is equivalent to defining a set in *list format* (but should not be confused with the Python `list` type). To check the size of a list or the cardinality of a set, use the `len()` function, as in `len(s)`.

Note you can also create a set from a list by using the `set()` function, passing the list as an argument.

Task: Try out the examples above, then modify the literal values assigned to `l` and `s` to be a list and, respectively, a set of:

- the first five positive numbers;
- five animal names (strings in Python can be any text within either single `'` or double `"` quotes, as long as they match).

The **`in`** operator is equivalent to $\in$, except that it works with any collection type in Python. Test the sets you just defined to see if they contain elements you know are (1) present and (2) not present.

3 Building sets and lists

In mathematics we can describe a set's contents by using a defining property (i.e., a predicate) that does not require us to explicitly list all its elements. In a *program* we typically need to *generate* the contents of a set, which we do by using the defining property. Some defining properties are more easily translated into a generating function than others.

One way to enumerate the elements of a list or set is to iterate over different values of some variable and apply the rule implied by the defining property. At this stage we'll look only at creating sets, but the code is almost identical for lists. For instance, to generate the set of squares of the first 10 positive integers (i.e., $\{1^2, 2^2, 3^2, \ldots, 10^2\}$) we could write:

```
s = set() #creates an empty set
for i in range(1, 11):
    s.add( i**2 )
```

where `range(1, 11)` creates a list of the values between 1 (inclusive) and 11 (exclusive, meaning the last value is 10), and `i**2` is the defining property of members of the set. From this example we can derive a general pattern:

```
s = set()
for i in range(low, high):
    s.add( generating_function(i) )
```

where you can change *low* and *high* as needed, and *generating_function* is a placeholder for whatever expression (i.e., rule) you need to generate the i^{th} element of the set. To generate a list instead of a set, simply replace the initial value of s with `[]` and the call to `add` with `append`.

Task: Using the template above, generate the following sets (create a new loop and set variable for each one so you can keep it as an example for later reference):

- multiples of 4 between 4 and 120 (i.e., {4, 8, 12, ..., 120})
- even numbers from 0 to 20 (i.e., {0, 2, 4, ..., 20})
- positive odd numbers up to 21 (i.e., {1, 3, 5, ..., 19}) [tip: try modifying your solution for the previous one]
- (optional challenge) the set {-27, -9, -3, -1, 1, 3, 9, 27}

Task: Inspect the value of the each set once it's been generated. If you like, create modified versions of your for loops to create lists instead of sets (change the initial value to `[]` and use `append` instead of `add`) and take a look at them. Do they appear to have the same order?

3.1 Set and list comprehensions

Python supports a compact way of generating lists, sets and dicts, known as *comprehensions*. Take a quick look at *The Python Tutorial*'s entry on list comprehensions (ignore the part that mention `map` and **`lambda`**, which we won't be doing yet, and stop before the longer examples). The syntax for a set comprehension is just like that for a list comprehension, except that the surrounding punctuation is `{}` instead of `[]`.

Using the template from above the general pattern is:

```
s = { generating_function(i) for i in range(low, high) }
```

so, for example, the squares of the first 10 positive integers would be:

```
s = { x**2 for x in range(1, 11) }
```

Task: Write set comprehensions for each of the sets you generated above. And also try writing them as list comprehensions. Are the results the same as before?

3.2 A final ~~improvement~~ complication

An alternative to using a generating function when you know the property that a set's elements must satisfy (and you also know the larger set of elements from which they are drawn) is to use an **`if`** inside the set comprehension, as in this example:

```
s = { x for x in range(0, 21) if x % 2 == 0 }
```

which iterates over values of x between 0 and 20 and selects only the even values (`%` is the modulo [remainder] operator in Python).

Summary

- sets and lists can be created by using a *generating function* over a list of values (the generating 'function' may be any suitable expression and doesn't have to be an actual function like $f(x)$);
- set (and list) comprehensions offer a compact way of describing the generation process;
- a conditional expression (`if`) can be used inside a set comprehension instead of (or in addition to) a generating function to select the subset of values that belong in the set.

Useful references

Spyder documentation
: https://pythonhosted.org/spyder

A bit about lists
: https://docs.python.org/3/tutorial/introduction.html#lists

Some built-in functions, including those for type conversion
: https://docs.python.org/3/library/functions.html

Python data structures
: https://docs.python.org/3/tutorial/datastructures.html

Notes

Division in programming languages

Some programming languages (e.g., C, C++, Java, and Python versions prior to 3) use the `/` operator to mean either integer division or real-valued division (also called 'floating point' division), depending on the type of the operands. So in those languages 1/2 is zero because 2 fits zero times into 1, but 1.0/2.0 is evaluated as 0.5.

Python 3, which we use in this unit, has two operators so you can choose which behaviour you want:

- `/` is the mathematical division you're used to: `1/2` evaluates to 0.5
- `//` means integer division: `1//2` evaluates to 0

About included solutions

To show available solutions in this and later labs click the Enable Solutions button. This will reveal Show Solution links that will show (or hide) solutions. Remember: attempt each task yourself first and only reveal the solution either to check your answer or if you have made a serious attempt and cannot work it out yourself.
