Lab 04: Functions and Dictionaries

Contents

Objectives

By the end of this lab you will understand:

- How functions avoid code reuse

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

- Define and apply functions
- Operate on dictionaries
- Engage with real datasets

1 Functions

1.1 Syntax

We have already seen that Python comes with many built-in functions, but you can also define your own. Using functions is an important way to avoid code replication when you want to repeatedly apply the same set of operations on different inputs, as you may have already encountered in previous labs. Defining your own functions is essential when you want to apply operations beyond the ones covered by Python's basic functionality (e.g., you want to measure specific properties of a specific dataset, as we do in this week's lab).

A *function* can be supplied *arguments* (inputs), and (depending on the function) produces a return value (an output); we have already seen this with built-in functions like `isinstance`, which takes as input two arguments, a value and a type, and returns a boolean. When defining your own function, you must specify not only the arguments but also the *body* of the function, which specifies the steps that are executed when the function is called. Syntactically, functions are created using the keyword `def`, as in the following example:

```
def addone(x):
    return x + 1
```

The variable `x` in the above is referred to as the *formal* argument. Executing the above two lines of code results in the variable `addone` being bound to a function value (with type `function`). It is not done as commonly as with other types, but you can assign this value to another variable in the usual way:

```
add1 = addone
```

1.2 How functions work

A function can have no arguments, one arguments, or many arguments. A function *application* is a kind of expression. It consists of two parts: an expression whose value is a function (typically a variable bound to a function), and a parenthesized series of expressions, separated by commas. For example, `addone(4+5)` is a function application.

To evaluate such expressions, Python first evaluates the expressions in the parentheses. In the example above, there is one such expression, `4+5`, and it evaluates to `9`. The formal argument is bound to this evaluated value (called the *actual* argument), meaning in the above example `x` is now temporarily bound to `9`. Next, Python executes the statements comprising the body of the function definition. If a `return` statement is executed, the corresponding expression is evaluated, and the resulting value is the *return value*, or output. If no `return` statement is encountered, no value is returned. (Technically, a value is returned of type `NoneType`.) In either case, the binding of the formal argument to the actual argument is deleted at the end, meaning `x` is still bound to the same value it was bound to before the function application (or is not bound to anything). Another way of saying this is that the formal argument is *local* to the function. Similarly, assigning to a variable within the body of a function doesn't change the bindings of any variables defined outside of the body. To exemplify this behavior, imagine we had instead defined `addone` as follows:

```
def addone(x):
    x = x + 1
```

Since this function has no output and integers are not mutable objects, the value of any input will be completely unchanged after applying `addone`. So, you would see:

```
>>> x = 9
>>> addone(x)
>>> x
9
```

If you instead tried to assign `x = addone(x)`, `x` would take on the value `None`, which again is the value returned by functions without an explicit `return` statement.

The only case in which functions should not have return values is then the case in which the arguments are mutable, in which case they can be modified within the function without being returned explicitly; this is just as we saw with built-in functions like `append`, where we write `mylist.append(1)` rather than `mylist = mylist.append(1)` (as again this would result in `mylist` taking on the `None` value). In general though, it is a good idea to make sure your functions have an explicit `return` statement along all possible branches.

1.3 Best practices

When defining a function, it is important to first write down (typically in a comment above the function definition) the function *specification*, which states what it does, in terms of its inputs and outputs, but not how it does it; this will be stated explicitly in the body. For example:

```
# ADDONE: adds 1 to a number
# input: a number x
# output: x + 1
def addone(x):
    return x + 1
```

1.4 Exercises

Task 1. Even though it's a built-in function, create a file `strings.py` and write in it a function `my_join` that has the same functionality as `join`; i.e., that given a separator `mysep` and a list `mylist`, outputs the same string as `mysep.join(mylist)`. For example, `my_join(' ',['a','b'])` should output `'a b'`. Test it by comparing its output to that of `join` on lists of strings. (Hint: iterate through `mylist` using a loop, but be careful with how you handle the last element, and how you handle the empty string as a separator.)

Solution 1. There are a number of ways to do this, and there are two fully commented and tested solutions in `strings.py`. In one of the solutions (`my_join2`), the function loops until the second-to-last element in the list, adding an element and the separator in each iteration, and then adds the last element manually. An alternative solution (in `my_join`) is to loop through the whole list, and then take the extra separator off at the end, as in:

```
def my_join(mysep,mylist):
    to_return = ''
    for x in mylist:
        to_return = to_return + x + mysep
    to_return = to_return[:-len(mysep)] if len(mysep) > 0 else to_return
    return to_return
```

Task 2. Add to `strings.py` and test a function `sort_string` that, using `sort`, `list`, and `join`, takes in a string and returns the alphabetically sorted version of that string.

Solution 2. The full commented and tested solution is in `strings.py`; here is the basic function definition:

```
def sort_string(mystr):
    mylist = list(mystr)
    mylist.sort()
    return ''.join(mylist)
```

Task 3. Use functions to revisit and improve on some of the work you've done in previous labs. In particular, copy over the `dna.py` file from Lab 02, and edit it to include functions. If you prefer you can also work with the solutions given rather than the one you wrote yourselves (and, in your own time, you might also find it helpful to go back and revisit other assignments, like `hangman.py` in Lab 03).

For `dna.py`, define three functions `reverse`, `complement`, and `reverse_complement` and use those to test out at least five different DNA strands (you need only give expected results for `reverse`, as the others are hardcoded). While you're at it, it should also be possible to use `if` statements to do complements in a better (but not necessarily shorter) way.

Solution 3. See `dna.py`. Because it is updated in Task 4, the solution in the file does not incorporate the `if` statements; this could be achieved using something like the following code snippet (which assumes `char` and `comp_char` have been defined):

```
if char == 'a':
    comp_char = 't'
elif char == 't':
    comp_char = 'a'
elif char == 'g':
    comp_char = 'c'
elif char == 'c':
    comp_char = 'g'
else:
    return 'error: non-dna character in string'
```

Although this solution is longer, it is better than the two strings used in the solution to Lab 02, as if those two strings become out of sync (for example, because the programmer has forgotten that they needed to be maintained together) then the program fails in a way that is more difficult to detect. In Task 4, however, we'll see a more elegant way to do this anyway.

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

2 Dictionaries

Thus far, we have done pretty well using lists as our only container type. One container type that is arguably even more useful, however, is a *dictionary*, which is used to map one set of objects (called the *keys*) to another (called the *values*). This means that in computer science dictionaries are also sometimes referred to as a key-value store.

Why is this so useful? Imagine that you are trying to keep track of a dataset, as you would in a spreadsheet. Using lists, you would likely end up using a list of lists, where each list represented a row in the spreadsheet. For example, imagine we stored survey results as lines in a CSV file, and recovered a single column representing age using something like the following:

```
columns = line.split(',')
age = columns[1]
```

This would require us to remember the order of the columns, since we are *hard-coding* age as the second column. Using a dictionary, however, we could map the semantic meaning of a column (name, age, etc.) to its value for that row, rather than using a number with no real meaning.

Syntactically, a dictionary can be created in a number of ways. To create a dictionary `secu2002` mapping `'lecturer'` to `'sarah'`, `'ta'` to `'enrico'`, and `'students'` to `['alice','bob']`, we could write

```
secu2002 = {'lecturer' : 'sarah', 'ta' : 'enrico', 'students' : ['alice','bob']}
```

Once we've created a dictionary, we can access its keys using the `keys()` function, and its values using `values()`, as in:

```
>>> secu2002.keys()
['lecturer', 'students', 'ta']
>>> secu2002.values()
['sarah', ['alice', 'bob'], 'enrico']
```

Assuming a key is in the dictionary (which we can check for using `k in secu2002.keys()`, or `k in secu2002` for short) we can access the value stored at that key using square brackets, as in:

```
>>> secu2002['lecturer']
'sarah'
```

We can also add values for new keys or replace the values at existing keys using an assignment, as in:

```
>>> secu2002['department'] = 'crime science'
```

This will overwrite the existing value stored at this key if it was already there, or create a new key-value mapping if it wasn't. Finally, if the value stored for a given key is a mutable type, like a list, we can update it, as in:

```
>>> secu2002['students'].append('charlie')
>>> secu2002['students']
['alice', 'bob', 'charlie']
```

While the values in a dictionary can be mutable, the keys must be immutable. You can see this for yourself if you try to use a mutable type like a list as a key, in which case you'll get `TypeError: unhashable type: 'list'`. It isn't important to understand why this doesn't work, except to say briefly that mutable types are incompatible with the hash-based representation used to achieve efficiency in Python dictionaries. (For an explanation of the abstract concept behind Python dictionaries, you're welcome to read further at: `en.wikipedia.org/wiki/Hash_table`.)

Task 4. Go back and rewrite the `complement` and `reverse_complement` functions in `dna.py` to represent the mappings between different bases using a dictionary. This should make your code much shorter than it was after completing Task 3.

Solution 4. The full solution is in `dna.py`, using a dictionary as follows:

```
base_mapping = {'a' : 't', 't' : 'a', 'g' : 'c', 'c' : 'g'}
```

3 Putting It All Together

With functions in place, you are finally ready to begin tackling the mess of real-world datasets, although we'll continue to improve on how we work with these datasets in the weeks to come. All datasets used in this module will be in the `data` subdirectory of the main git repository, and the one we'll be using this week is `church_metal_theft.csv`, which is a CSV file documenting 130 thefts of metal from churches in a northern English city between 2009 and 2013. To protect anonymity, all personally identifying descriptions (primarily names and numbers) have been replaced with XYZXYZ.

The file contains five columns: the first gives the date that the crime started (in dd/mm/yyyy format); the second the time (in HH:MM:SS format, using 24-hour time); the third the date the crime ended; the fourth the time it ended (with these columns in the same format as the first and second, respectively); and the fifth a free text field containing a description of the crime. Because the free text field contains both commas and tabs, the column delimiter is `:::`.

Task 5. Create a file `church_theft.py`. First, write code in this file to read in the dataset and parse it into a spreadsheet format, either as a list of lists or as a list of dictionaries. Each element in the list should represent one row in the spreadsheet.

It is often theorized that crime follows seasonal patterns [1, 2], so fewer crimes are committed in the winter months and more in the summer months. To test this out, go through the data and tally the number of crimes committed per month and the number committed per season (treating December to February as winter, March to May as spring, June to August as summer, and September to November as autumn). For each, you should print out: the maximum number of crimes committed, the season/month(s) in which that occurred, the minimum number of crimes committed, and the season/month(s) in which that occurred. You will earn marks based not only of the correctness of your answers to these specific questions but also on the elegance of your solution. In particular, you can earn one bonus point by printing out months as their words (e.g., January) rather than their numeric values (e.g., 01).

Solution 5. The full solution is in `church_theft.py`. To check your answers, the maximum number of crimes in a season is 38 (in spring), and the minimum is 23 (in winter). Per month, the maximum number is 19 (in April) and the minimum is 5 (in both June and December).

4 Handing In

As always, feel free to continue trying out the various concepts we've covered in this lab. Once you are done, add all relevant files into a `lab04` directory; this should mean adding:

- `strings.py`
- `dna.py`
- `church_theft.py`

Push these files, along with the usual `partner.txt`, to the remote repository, and you're done! Please check with one of us on your way out.

References

[1] Graham Farrell and Ken Pease, Crime seasonality: domestic disputes and residential burglary in Merseyside 1988-90. *British Journal of Criminology*, 34(4):487–498, 1994.

[2] Martin A. Andresen and Nicolas Malleson, Crime seasonality and its variations across space. *Applied Geography*, 43:25–35, 2013.