COMP1002 Data Structures and Algorithms

Python for DSA – A Quick Guide v2.0

Note: These are a volunteer-produced resource by fellow Curtin students aimed to help you revise. They are not a substitute for lecture notes, other unit materials or advice from the unit coordinator. These notes were significantly updated in December 2023.

1 Introduction

In Data Structures and Algorithms, you have the choice of using either Java or Python. This guide attempts to cover the key background Python knowledge you will need in order to work effectively in DSA. The bibliography includes references to books that past students have found useful – Goodrich and Lee have excellent explanations of some concepts in this unit, while the others are good Python resources more generally.

`var` where it appears in this guide just means "any variable".

2 Some warnings

In DSA, the following **must not be used** when coding in Python. There are significant penalties if these are found in any pieces of assessment, and especially the Final Assessment:

- **Lists**. Identifiable by use of `var = []` or `var.append()` amongst other constructs.
- **Dictionaries or sets**. Identified by use of {} characters in the declaration, or `set()`.
- Any inbuilt or library-imported structures or sorts, other than when you are *explicitly* told otherwise (i.e. in the final lecture and any exam questions which come from that lecture).
- Any special methods in numpy, other than those needed to create an array.

In place of lists, you can use either an numpy (np) array or your Week 4 linked list (or any other structure you have created in pracs). In place of dictionaries, you can use your Week 7 hash table. You *may* use display libraries such as networkx that do not affect processing of the code in your assignment, but *check first* (we suggest asking on Piazza.)

We cannot stress this enough. We know of fellow students who have lost huge amounts of marks for ignoring this advice, that is why we are putting it on the front page. It does not matter that many online sources and even some of the books we have recommended use these things (especially lists) – you cannot in this unit.

If you have any concerns or need any help in using the correct structures, please ask on Piazza, ask your prac tutors or come to Senior Tutors room when sessions are on.

3 Python syntax refresher

Especially if you haven't written Python code for a while or have been coding in other languages, here is a refresher on basic Python syntax:

- Printing is simple – `print("this text")` to print "*this text*" to the screen, `print(a)` to print the contents of a, and `print(a, b, c)` to print the contents of three variables with spaces between them.

- Python relies on indenting to decide at which level to run a line of code. Indents are a multiple of four spaces in Curtin units. If you get an "indentation error", make sure there are no tab characters mixed in and that there isn't an extra or missing space somewhere.

- For and while loops, if statements and related elif and else clauses, function definitions and a few other constructs must all end in `:`. e.g. `if 1 != 0:`. Subsequent indented lines tell Python that they are part of the above construct.

- Especially for those used to C or Java – in order to cycle through a list of numbers, Python syntax is `for i in range(max):`. The standard `for i in thing:` is actually an object iterator, like the "foreach" construct in Java.

- Common error – don't forget `=` is the assignment operator (left takes on the value of right), whereas `==` is the equality comparison operator (does left equal right? If so, output True)

- Booleans in Python are `True` or `False` - make sure to get the capitalisation correct.

4 numpy arrays

An array is a named list of data items that all have the same data type. Each data item is an *element* of the array. It has a fixed size and is located in a contiguous section of memory which is allocated at the time it is created. To create an array, make sure you have the following line at the top of your code:

```
import numpy as np
```

This imports the numpy library which gives Python the capacity to handle arrays. To create an array, there are a couple of options:

```
myArray = np.zeros(size, dtype=object)
myArray = np.array(list) # for converting a list
```

`size` is a number, which may be a constant or a variable. As arrays cannot be resized, this is a hard maximum for the array's size. (**Do not** use `myArray = [None] * size` - that is a list! Use one of the above only.)

Some useful commands:

- `len(myArray)` returns the length.
- `myarray[a]` returns element a from the array. Note that these are zero-indexed - the first element is 0, the second element is 1, etc.

5 File handling

Opening a file can be done in one of two ways:

1. The "equals open" method, which is useful if you need to keep a file open over a period of time and a number of operations. It is formatted:

   ```
   f = open("abc.csv", "r")
   ```

   You will need to close the file using `f.close()` once you are done. Be careful to ensure that both are executed in the same scope!

2. The "with open" method, which is useful if you need a file open for just enough to either grab its contents or write to it. It is formatted:

   ```
   with open("abc.csv", "r") as f:
       lines = f.readlines()       # example operation
   ```

   Once the indent drops back, the file automatically closes.

Reading from a CSV or text file usually looks something like the example below:

```
with open("abc.csv", "r") as f:
    lines = f.readlines()

for line in lines:
    var = line.strip().split(",")
    # add the elements to your structure (queue, tree etc) here
```

Writing to a CSV or text file is much more context-based and less standard, but here is one example which shows the write command in action:

```
with open("def.csv, "w") as o:
    for ii in structure:
        output = # assemble your output string
        output += "\n"
        o.write(output)
```

Note that there is no "writeline" command, and write does not add a new line character by default, so your code has to do that. The "strip" part removes white space and the new line character from the input line.

6 Handling objects

Note: This assumes you know the basics of object orientation, and is more focused on the Python implementation details.

Creating classes

When creating a class, you need a class definition, and a `__init__` constructor method. The constructor method, and in fact all class methods, will refer heavily to `self` – think of `self` as being the *instance created by the constructor* rather than the class itself, even though the code is being written in the class. The class is just a template for producing instances. So if you have a Car class, `self` refers to *a specific car*.

Thus, when you bring in input variables in the constructor method, they have to be converted to instance variables for each instance.

```
class Car:
    def __init__(self, model, colour, numberplate, kms):
        self._model = model
        self._colour = colour
        self._numberplate = numberplate
        self._kms = kms
```

So a statement `myCar = Car("Camry", "blue", "1ABC 123", 185421)` in our extremely trivial example uses the constructor method to produce an instance of class Car with those attributes.

Let's write a method. Note that if the method has no inputs, we still need to give it the `self` context.

```
    def display(self):
        print("I am a", self._colour, self._model, \
              "with numberplate", self._numberplate)
```

(That backslash allows us to run the command across two lines.)

So we run `myCar.display()` (the empty brackets indicate it is a method with no inputs) and it returns with:

```
I am a blue Camry with numberplate 1ABC 123
```

Inheritance

Inheritance ("is-a" relationship) allows us to work with sub-types. Let's say we want to have Sedans and SUVs, each inheriting from Car but with a couple of extras.

```
class SUV(Car):       # this means "inherit from Car"
    def __init__(self, model, colour, numberplate, kms, fuelEconomy):
        super().__init__(model, colour, numberplate, kms)
        self._fuelEconomy = fuelEconomy
```

This looks pretty mad, so let's break it down. `super()` is the parent class. This one line of code loads the parent class's constructor method into an instance of the child class, grabbing the parent's class variables and methods along the way. Note that you only need to separately add the variables that the parent class doesn't have. A common error – do not put "self" in the `super()` line.

One key feature of inheritance is *polymorphism*. What this means in practice is that while some methods are common to all (think `.isEmpty()` in some of the DSA pracs), some of the methods in the parent class may end up looking very different depending on which child class uses them.

So in the parent (Car), we remove our code for display, and change it to the keyword `pass`, which means "do nothing." [7]

```
# in original Car class
def display(self):
    pass

# in new SUV class
def display(self):
    print("I am a", self._colour, self._model, \
          "with numberplate", self._numberplate)
    if self._fuelEconomy > 12:
        print("My fuel economy isn't great.")
```

So the method in Car is *there*, but it ends up getting overwritten by the child classes which can implement it differently.

Aggregation

Aggregation is more subtle, but leads to some complications. The main thing you need to be aware of is what *scope* you are currently looking at. The most common result of not getting this right is strange errors like `AttributeError: 'str' object has no attribute 'head'`.

In the graphs prac, for example, where one of the graph's class variables is a linked list, this is what you may have to consider:

- `self` is the graph.
- `self.vertices` is the linked list of graph vertices.
- `self.vertices.head` is the head node of that list.
- `self.vertices.head.value` is the object inside the head node, which may itself have a property (hence another dot and another suffix) which you may need to access.

You can simplify this by having, e.g. `currNode = self.`<...>, but even then, you still may end up looking at the wrong level.

Our suggestion is to map these things out with pen and paper so you always know where you should be, and where you are. Use `print` statements at key points to test what you're looking at (e.g. `print(type(currNode))`).

7 Errors and exceptions

This section is often not done correctly in DSA practicals. The goal is to ensure your program handles exceptions – whether system-generated or generated by your code – gracefully.

When working in a single block of code, if you raise an exception, you need to catch and handle it as well. For example in the factorial code:

```
    try:
        if n < 0:
            raise ValueError("Input must be a positive integer!")
    except ValueError as err:
        print("Error: ", err)
```

If you do not run the `raise` command inside a try/except block, the code simply crashes, displaying your error.

However, when the code is inside a class that you have imported (e.g. DSALinkedList), you actually *don't* want it to be handled by the class. If that error is occurring, you want to send it back to the calling code. So for example in a pop() method in a Stack class, you might raise an error:

```
def pop(self):
    if self.isEmpty():
        raise StackUnderflowError("Stack is already empty")
```

And then in the calling code, you would have this:

```
    try:
        value = stack.pop()
    except StackUnderflowError as err:
        print("Error: ", err)
```

And in your test harness:

```
    # pop from an empty stack - should have error
    try:
        value = stack.pop()
    except StackUnderflowError as err:
        print("Test passed!")
```

A *big* no-no is to `raise Exception`. Exception covers all possible errors. Let's say you raise one, and then your handler `except Exception` picks it up. How do we know it was your error and not some other random system error?

Page 18 (current as of 2023) of the DSA Lecture 2 notes [6] provides code which can be used to create and name your own error classes. We strongly suggest adapting and using this, especially when it comes to your assignment. Ask the tutors if unsure how to apply it.

8 Serialisation

Let's say you wanted to create a linked list of graph nodes, or some other structure. The inherent hierarchy and relationships between these objects, as well as properties and internal states of the objects themselves, is what makes this structure work. Up until now, methods you have learned to store such objects in a file actually remove all of this key information - what is in the file are not objects, but their inputs and outputs. CSV or text is a "flat file". This means that if you want to recreate the objects later, you need to bring in the text file, then go through each item and create it from scratch, and reassemble the overall structure. This increases the chances of errors, or of two users of the same information having different structures or results.

What if you had a way to just "unplug" the structure, put it away in a file and then "replug it in" with only a couple of lines of code? This is the magic of serialisation - it stores the objects in a machine-readable (but not human-readable) format which preserves the hierarchy and relationships and properties that are lost when we write to CSV or text.

Here's how you do it:

```
# import the pickle library
import pickle

# storing an object using pickle
# don't use .txt or .csv as the filename - it will confuse you later!

with open('myObject.ser', 'wb') as f:
    pickle.dump(ll, f)    # ll can be any object name

# getting back an object using pickle
ll = DSALinkedList()
with open('myObject.ser', 'rb') as f:
    ll = pickle.load(f)
```

A couple of points: firstly, the "wb" and "rb" are necessary because we are reading or writing to a binary file, not a text file. And secondly, when recreating the object, Python needs to know what socket on the wall to plug it into. Is it a linked list? Is it a graph? Or something else? So you need to set up a blank structure of the right type, and then Pickle will fill in the details when it loads your object.

9 Odds and ends

If you get an error involving NoneType when invoking a function, check to make sure you remembered to return a value from the function.

Occasionally it is helpful to force a data type in a function header:

```
def aFunction(b: int, c: str) -> DSAListNode:
```

This translates to "Input variable b must be an integer, input variable c must be a string, and the output must be a DSAListNode". Note that this header does not actually convert – you have to do that, or a TypeError will occur.

References

1. Barry, P. (2017). *Head first Python* (2nd ed.). O'Reilly Media. ISBN 978-1-49191-953-8.
2. Downey, A. (2015). *Think Python: how to think like a computer scientist.* (2nd ed.) Green Tea Press. https://greenteapress.com/wp/think-python-2e/
3. Gaddis, T. (2021). *Starting out with Python* (5th ed.). Pearson. ISBN 978-0-135-92903-2.
4. Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2013). *Data Structures & Algorithms in Python.* John Wiley and Sons. ISBN 978-1-118-29027-9.
5. Lee, K. D, & Hubbard, S. (2015). *Data Structures and Algorithms with Python.* Springer. ISBN 978-3-319-13071-2. Available online through Curtin Library.
6. Maxville, V. (2022). *Data Structures and Algorithms* [PDF slides]. Curtin University.
7. W3Schools. (2020). Python inheritance. `https://www.w3schools.com/python/python_inheritance.asp`