CS: Programming Techniques Notes

Overview

Programming techniques is one of the four main sections of the controlled assessment. It is the one that is awarded the most marks (six times more than any other section) and so is the most important.

What are the marks for?

There are 36 marks available for this sections, split into six six-mark categories:

- Interoperability - How do the program parts *fit together?*
- Working - Does your program *work?*
- Efficient - Have you used *efficient techniques?*
- Data Structures - *What* and *why?*
- Robust - Does your program *survive* the user?
- Understood - Do you *understand* what your doing?

Marking Levels

There are three levels of marking for each category - you can and should be aiming for the highest one:

- State - Say *what* you have done.
- Explain - Say *how* it works.
- **Discuss** - ***Why*** this way?

In order to achieve level two, you must also achieve level one, and for three, two, etc.

Structuring This Section

I would suggest splitting your answer into categories and then discussing individual techniques within categories. There are six points you want to make for *each* technique:

- What - What is this? Give a brief introduction.

- Why
  - Alternatives - What else could you have done? Arguments both for and against all alternatives.
  - Justify - Why did you choose the method you chose? Why is it *better*, not just *good*?
- Example
  - Where - When have you used this? What did you do?
  - Context - Why is it *justified* in *this* case?
- Explain - *How* does this technique *work?* Show your *understanding.*

Aim for one point, per category, per technique. So ideally, you would have six techniques in each category, although this may not be achievable, it may all balance out - some sections have lots of techniques, others have only two or three.

What are programming techniques?

It may seem difficult to find programming techniques to discuss, but almost *anything* can be considered a programming techniques. If you use something in your program even once, *include it!* Here are a few things that could be considered programming techniques:

- Built in Python functions - `map`, `filter`, `char`, `random.randint`, etc.
- Classes
- Functions
- Loops - `for` and `while`.
- Variables - also `global` and `nonlocal`.
- Lists - also tuples and dictionaries.
- Strings
- Files
- Any other data types.
- Any patterns you define - if you do something in a particular order or way, use that too.

Analysis of the Categories

Working

This is six easy marks, simply have a working program. The evidence for this is in the *testing* section. Specifically, the evidence is *passed* tests that use *valid* input. What I mean is tests in which you *didn't* try to break the program, and it *worked as expected.* That demonstrates that the program is working.

Interoperability

For this, I recommend dividing your program up into units. If you have functions or classes, this is a good place to start. For instance, in my last controlled assessment, I had thirty functions. Your program probably won't have that many, but the theory is still the same. A diagram might be good here, with some description. The main point is that you must say *what* provides *what* to *what* for *what* purpose.

Efficient

Efficiency takes many forms - speed of your program, the memory requirements, the size of your program's code, the readability and ease of understanding of your code, etc. The key here is *discussion. Why* are the techniques you have chosen *better*? Here, it is crucial to provide alternatives.

Robustness

- Robustness is the ability of your program to *survive testing.*
- For tests which you tried to break the program and failed, how did it survive?
- For tests which *did* break the program, why did it break and how are you going to fix it?
- It's not about your program crashing of its own accord; It's about the *user's input* crashing your program.
- Tests in which you don't try to break your program don't prove it is robust - simply that it works.

Data Structures

- A *program* is a set of *algorithms* which convert between *data structures.*
- A data structure is a way of storing and representing data.
- *Where* do you store data and *why?*
- The key here is also *discussion* - you must provide alternatives.
- Variables, lists, strings, numbers, *why did you use those?*
- Include any *data types* you used.

Understood

- ***How*** does it work?
- ***Why*** does it work?
- ***Why*** like that? Offer alternatives and then justify your *choice.*
- ***Where*** does it do it like that - give examples.
- ***What*** does it do?
- Show your *understanding.*

An Example

Programming Techniques Used

Lists

I use lists throughout my program as a way of storing a variable quantity of data. There are several alternatives I could have used instead of lists. For example, I could have used many variable declarations. However, this isn't usable since I need to hold a varying quantity of data. Also, and extension that would allow we to do this with variable definitions is much less efficient and much more unsafe, and less reliable. Another alternative could have been using a tuple. Instead of the normal list syntax:

```
list = [1, 2, 3]
```

Tuples use the alternate syntax:

```
tuple = (1, 2, 3)
```

There are several arguments for using tuples. The first of these is that creating tuples is more memory efficient and faster. To show you why I used lists instead, I will take an example of the word lists in my program:

```
with open('words.txt', 'r') as f:
    return list(f.read().split('\n'))
```

Here, I used a list because I needed to modify the list later:

```
words[-1], removed = replaced, words[-1]
random.shuffle(words)
```

If I had used a tuple, since they are immutable, whenever I modify it, I would have to create a new tuple. This is slow and memory inefficient because tuples do not allow in-place modification. However, wherever I am using lists which I do not modify, I have used tuple as they are more efficient to create. Another reason I have used lists is that it is more readable and easy to understand to use the list methods:

```
list.append(item)
list.remove(item)
list.index(item)
```

Rather than the tuple equivalent:

```
tuple += (item,)
tuple = tuple[:tuple.index(item)] + tuple[tuple.index(item) + 1:]
tuple.index(item)
```

Another alternative is an iterator. However, this is not very useful, since iterators are not modifiable, and they cannot be indexed (elements cannot be selected at random, only in order):

```
iterator[index] #this doesn't work
```

Also, you can only go through the elements of an iterator once. This means that I would have to save them to a list, thus negating any possible benefit. Despite this, iterators are used occasionally when all these requirements are met, since they are much more efficient than either lists or tuples, only storing one element in memory at a time.