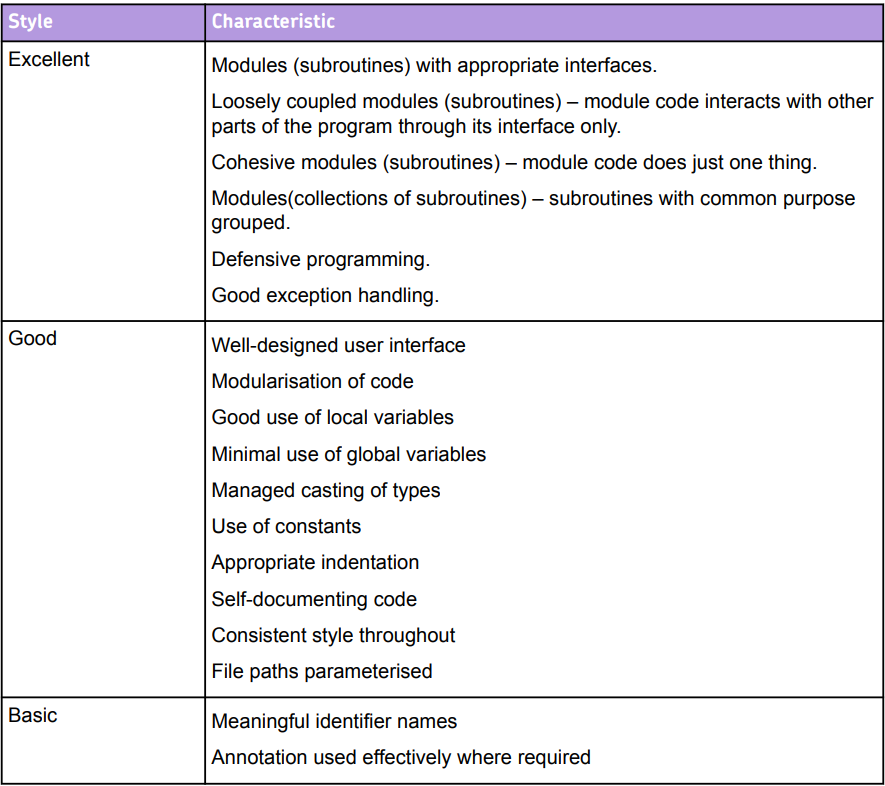
Guide to commenting and other cool stuff



Code comments

Commenting your code is a requirement to even meet the basic level for coding style on the NEA. However, it is important to remember your audience. The people who would reasonably be reading your code would be yourself and other software developers. This means that you do not need to write line by line comments for basic code structures that any developer would be able to understand without the comments. Where we do most of our commenting is at the Class level and at the function level.

For each class you need to produce a docstring which will contain the name of the class and a description of the purpose of the class. E.g.

*'''  
Name: Treasure  
Purpose: This is to create treasure objects which can  
be hidden on the grid for players to search for  
in the game  
'''*class Treasure:

Then at the function level we ned to also include parameters and returns along with their data types. Parameters being the values passed into a function and returns being the value/values returned upon completion. E.g.

*'''  
Name: \_\_init\_\_  
Parameters: value: integer, level:string  
Returns: None  
Purpose: Constructor to set the initial values  
of the treasure object  
'''*def \_\_init\_\_(self, value, level):  
 self.\_\_value = value  
 self.\_\_level = level

Where we do produce more detailed comments will be in any bespoke, complex functions you have created which may require a little more explanation. It will be down to your judgement as to which functions require this but everybody should have at least a few which are not standard.

Excellent coding style

Here I am going to talk about a couple of key things that fall under the “Excellent” banner of the coding style for your techniques used. It is important to note that it is perfectly possible to achieve 27 out of 27 without doing either of these is your system is very complex. These are however, very good practice and a relatively easy way to pick up those Group A marks compared to the level of complexity you need to reach without them.

Exception handling

One of the things listing under excellent coding style is “Exception handling”. This is where you build safeguards into your code so that instead of crashing it will catch the exception and do something else instead. This is generally very useful around user inputs and most bugs in a system will arise from users doing something stupid.

This will usually be done through a try-except statement. With the following code, without the try-except the program would crash if I were to enter a string as it cannot be cast to an integer. In this case the ValueError exception would be caught and the message would be printed and a crash prevented. There are different types of errors that you can check for in a try-except statement and produce different results from each type of error. You can also raise a general exception that will trigger from any potential crash you haven’t already accounted for.

num= 0  
try:  
 num = int(input("Enter a number"))  
except ValueError:  
 print("Please enter an integer")

Defensive programming

If exception handling is catching errors as they happen to prevent a crash, defensive program is building your program in such a way that the crash is never at risk of happening in the first place. One of the key ways that we can do this (and demonstrate excellent understanding of OOP at the same time) is to make correct use of encapsulation. This is where all of our attributes are made private and can therefore only be accessed and modified through public getter and setter methods. Consider the following code:

class Treasure:  
  
 def \_\_init\_\_(self,value, level):  
 self.value = value  
 self.level = level  
  
t1 = Treasure(35, "C")  
print(t1.level)  
t1.level = "B"  
print(t1.level)

Here I have not used the principles of encapsulation. I can both view and change the attribute of level from outside of the class. This is considered bad practice as unintended changes can be made to your attributes which is against the principles of OOP. So instead I’m going to make them private and we can access them through getter and setter methods. These are special public methods created for each of your private attributes that need to be accessed outside of the class. I’m going to show you two ways of doing this: this first is the more traditional way that you would use in many programming languages such as Java or C# and the second is the “pythonic” way of doing it. Either method will gain you the same amount of credit so your decision will come down to personal preference.

Traditional:

class Treasure:  
def \_\_init\_\_(self, value, level):  
 self.\_\_value = value  
 self.\_\_level = level  
  
 def get\_level(self): return self.\_\_level  
  
 def set\_level(self, level):  
 if level == "C" or level == "S" or level == "G":  
 self.\_\_level = level  
 else:  
 print("Invalid level")

t1 = Treasure(35, "C")  
print(t1.get\_level())  
t1.set\_level("S")  
print(t1.get\_level())

Here I have created my getter and setter for the private level attribute. The getter simply returns the private level attribute so that I can print it or use it in some other way. The setter is used to modify the value. It is in the setter that we can see the demonstration of encapsulation and therefore defensive programming. The only way to modify self.\_\_level is through this setter method and we can see that the level is only updated is the programmer passes in one of the three valid letters, anything else and the program will reject it. Note that we can combine this with exception handling and where appropriate we can build a try-except inside a setter.

To view and update the value I have to call the getter and setter methods for the object.

Pythonic:

class Treasure:  
def \_\_init\_\_(self, value, level):  
 self.\_\_value = value  
 self.\_\_level = level  
  
@property  
 def level(self):  
 return self.\_\_level  
  
 @level.setter  
 def level(self, level):  
 if level == "C" or level == "S" or level == "G":  
 self.\_\_level = level  
 else:  
 print("Invalid level")  
  
t1 = Treasure(35, "C")  
print(t1.level)  
t1.level = "S"  
print(t1.level)

Here instead of creating methods called “get\_level” and “set\_level” I have used something known as decorators which are a feature of Python that many other languages lack. @property is used for the getter (and must be declared before the setter) while @level.setter is used for the setter (level because it is the name we want to reference our self.\_\_level attribute) and both methods are now called level. If we look outside of the class we can see that I print(t1.level), to an untrained observer it looks like we are just printing a private attribute but we aren’t. The private attribute is \_\_level rather than level, when we write this code python figures out that it needs to run the getter and so will return the private attribute that way.

Similarly with the setter we can seemingly just changed the value of the private attribute but we aren’t. When I write t1.level = “S” it runs the setter method and passes in “S” as the parameter.