COMP1005 Week 7 Cheat Sheet

Lisa Luff

10/6/2020

Contents

My Own Info

Pass

- If you want to do diagnostics, and have initialised or partically completed the code, you can write pass to pass it by, rather than have an error

- In code:

if item in thing == T:
- print(“thing”)
elif item in thing == F:
- pass
else:
- pass

- This means that if you just want to test if it prints when T, but haven't written what to do otherwise yet, you can just put in pass, and python wont put out an error when trying to run it for having empty statements

Self

- When you create a new class instance, the variable assigned to it automatically is made the self value.
- This means that anything for the specific instance should start with self
- Python will then use self as the variable name, or vice versa
  - So in your code, in the init function, when you write self.name = name, you are telling it that the value for name in that specific instance (variable), is the value for name given in the class arguments
    - So what is actually written is *variable*.name = name
  - It also means that in any methods, or when you call on class methods, self will automatically be changed to *variable* and anything done by or with the method which starts with self, will be done for the specific instance *variable*

Muting

- If something is mutable, it means it can be changed
- If something is immutable, it means it cannot be changed
- Mutable -
  - Things like lists can be changed dynamically, in classes too
  - This means that you can change any part, at any time
    - In classes, it means for any instance, you can change the values you already assigned it at any time, including global variables for the class by calling on that variable for that instance
- Immutable -
  - Strings and tuples cannot be changed dynamically
  - This means that you cannot change one part of it at a time, only completely rewrite the variable it's contained in
    - You can however slice it, replace the part you want to change, and then put it back together again
      - This cannot be done with things like the append function though

Dunder Methods

- Dunder methods are functions in class with double underscores either side
- They are used to customise classes
- Important but more advanced
- Examples:
  - ___init___ - to define what is in a class
  - ___str___ - to define what to print if you print an object (have to press F5 before use)
  - ___main___ - to prevent the main chunk of a program running if you just want a function used in it

Object Relationships

Goals of OO restated in technical terms:

- Reuse/Extensibility
  - Reuse: Each class provides its functionality to other classes
  - Can inherit functionality from another class to reuse and then extend that other classes functionality
- Modularistion
  - Low coupling: The user should only ever need to work with/worry about the user interface, and never the backend/details of how it works (is implemented)
  - High cohesion: A class should only contain classes and modules relevent to a single purpose/task. Additionally, all classes and modules related to that purpose/task should be held within that class.

Class Relationships

- When classes have an association, they communicate with each other via message passing
- Classes that do this are said to have a relationship
- There are a number of relationship types, depending on how that message passing occurs
- Relationship types:
  - Aggregation
  - Composition
  - Inheritance
  - Other

Aggregation

- One class is part of the other, but both can exist separately
  - Eg, An employee is part of a company, but if the company shuts down, the employee still exists, just not as employee of that company
- As such usually one class, will be used as a class field of the other.
  - Eg, The information from the employee class is useful for the company class, and so the company class will call on the employee class (within the class field of the company class)
- This means that both classes are written separately

- There is a level of one-way ownership (or inheritance)
  - Eg, An employee can belong to the company, but the company cannot belong to the employee (owners don't count as employees)

- Think of it as class type **has a** other class type
  - Eg, A company *has an* employee
- In code:

```
class Employee():

- listofemployees = []

- def ___init___(self, name, company, department, role):
-- self.name = name
-- self.company = company
-- self.department = department
-- self.role = role
-- listofemployees.append(self.name, self.company, self.department, self.role)
-- return(listofemployees)

class Company():

- def ___init___(self, name, purpose, worth):
-- self.name = name
-- self.purpose = purpose
-- self.worth = worth

- def companyemployees(self, name, department, role):
-- self.companyemployees = Employee(name, self.name, department, role)
-- return(self.companyemployees)
```

Composition

- One class is exclusively part of another class, and that other class cannot exist without the main class
  - Eg, A company has many departments with different roles within the company. However, if the company were to shut down, the various departments would no longer exist.
- This means that one class exists within the other.

- Again this is a **has a** relationships, where one class *has a* other type of class
  - Eg, Company class *has a* type of department class

- In code:

class Company():

- def ___init___(self, name, purpose, worth):
-- self.name = name
-- self.purpose = purpose
-- self.worth = worth

- def companyemployees(self, name, department, role):
-- self.companyemployees = Employee(name, self.name, department, role)
-- return(self.companyemployees)

- class Department():

-- def ___init___(self, name, purpose):
--- self.department = name
--- self.purpose = purpose

-- def departmentemployees(self, name, role):
--- self.departmentemployees = [] --- self.employees = companyemployees(self, name, self.department, role)
--- self.department = [self.departmentemployees.append(item) for item in self.employees if department == self.department] --- return(self.departmentemployees)

Inheritance

- One class decends from another class
  - **Child/sub-type/derived** - is the decendent
  - **Parent/super-type/base** - is the parent
    - Eg, For the company class, that company might have smaller companies that are part of that company, who you want to do the same things as the larger company, but also say something about the larger company it is a part of.
- The decendent, or child, inherits (or has the same) attributes and methods as the parent class, but it can change them (override them), or add to them (extend)
  - Eg, You inherit thin brown hair from your mother who has thin brown hair. However you can override the colour by changing it to purple with dye. Or you can extend your hair to have more hairs, by adding to it with extensions.
- You don't need to have the children classes within the parent function, just within the same file
- This is a **is a** relationship, where one class *is a* child or parent of the other
  - Eg, Company *is a* parent to its subsidiary. Subsidiary *is a* child of the company
- The children inherit the attributes and methods as the parent by having the parent within the parenthese of the child class
- To overwrite the method or information from a parent class in a child class, just call it the same name
- Using super():
  - super() when used by itself is a call to the parent class ___init___ function
  - super().method() is a call to a method in the parent class
- In code:

Parent
class Company():

- def ___init___(self, name, purpose, worth):
-- self.name = name
-- self.purpose = purpose
-- self.worth = worth

Child
class Subsidiary(Company):

- def ParentCompany(self, name):
-- self.parentname = name
-- print(self.name, "is a part of" self.parentname)
-- return(self.parentname)

- def companyemployees(self, *listofemployees* = self.parentname.companyemployees)
-- return super().companyemployees(self.companyemployees)

Other

- The objects of one class are related to another in a manner that is not one of the above
- Will be discussed in future units

The Base Class

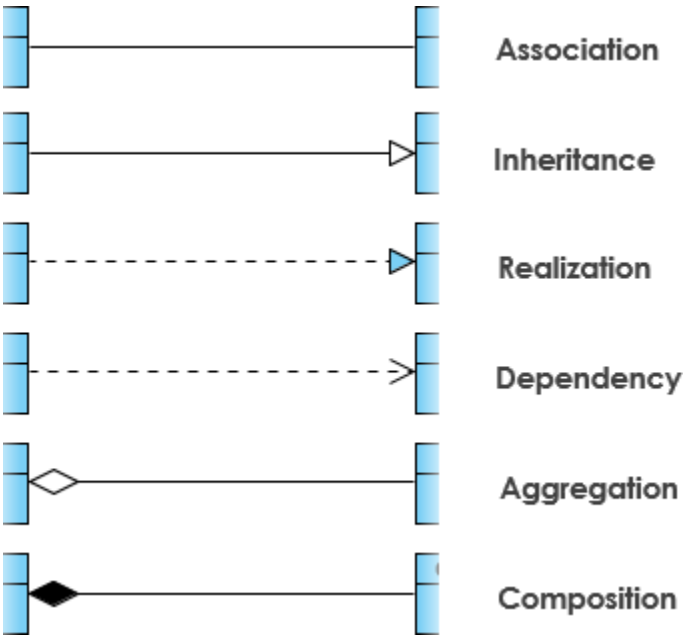
- All classes except for the base class, inherit from the base class
- The base class is *object*

Multiple Inheritance

- One class can inherit from more than one parent
- You do need to be careful that things that are named the same in both parents don't get confused in the child

Modelling Languages

- Models represent the relationships between classes, and instances of classes
- Usually a graphical image
- Most common method is **UML**:
  - Unified: Union of the approaches by Grady Booch, James Rumbaugh and Ivar Jacobson
  - Modelling: A graphical representation (or model) of an OO software design
  - Language: A standard way of expressing object relationships (syntax, and semantics)
- Relationship representations:
  - Aggregation - Open/unfilled diamond beside container class
  - Compostition - Solid diamond beside container class
  - Inheritance - Open triangle arrowhead beside parent
- Non-relationship representations:
  - Shows that one class invokes the method of another, but otherwise the relationship is unspecified
  - Association - solid line
  - Dependecy - dashed line
  - With arrow - unidirectional (one calls on the other but not vice versa)
  - Without arrow - bidirectional (they both call on each other)



Errors

- Errors and mistakes are often called bugs
- Main types of errors:
  - Syntax
  - Runtime
  - Logical

Syntax Errors

- Will cause the program to exit with an error message and wont run anything
- These are mistakes with spelling, or python "grammar"
- Examples:
  - Forget a keyword
  - Put a keyword in the wrong place
  - Left out a symbol
  - Misspelled a word
  - Incorrect indentation
  - Empty block

Runtime Errors

- Will run the program up until it hits a critical error, at which point the program will crash
- Examples:
  - Division by zero
  - Performing an operation on an incompatible type
  - Using an identifier that has not been defined
  - Accessing a list element, dictionary value, or object attribute that doesn't exist
  - Trying to access a file that doesn't exist

Logical Errors

- The program will run with no errors, but the output will be incorrect
- This is an error in the logic of the way the code has been written
- Examples:
  - Using the wrong variable name
  - Indenting a block incorrectly
  - Using integer division instead of float division
  - Having operator precedence wrong
  - A mistake with a boolean expression
  - off-by-one, and other numerical errors

Exceptions

- The OO way of handling errors
  - Handles errors more elegantly
- How it works:
  - Allows your program to handle errors by using methods to "catch" an error when it occurs, and then "throws" an exception
  - The exception is looking for a specific type of error, and will do something else if an error is caught, rather than not running the program, it crashing or giving an incorrect output
  - If the error is uncaught, the program will just handle the error itself in the usual way
- Python has a range of exception classes that all decend from the Exception class
  - You can define your eown exception class, so long as it inherits from the Exception class, or one of its children

Errors vs Exceptions

- When something goes wrong as it goes wrong, it is said that an exception is "thrown"
- If the exception is "caught", the method specified in the case of the issue will handle it. In this case, nothing will happen outside what has been specified in your program and the issue will remain an exception.
- If the "thrown" exception is left uncaught, when it causes the program to not run, or crash, etc. it becomes an error

Catching Exceptions

- try: Define the set of statements whose exceptions will be handled by the catch block of code that goes with this try
- except: What to do if an exception is throw in the try
- else: What to do if it was something you didn't specify. Need to put as *something* for exceptions specified separate to else
- finally: processing to always do regardless of whether an exception occurs or not

- In code:

try:
- number = int(input("Enter a number:"))
except ValueError:
- print("Input need to be a number.")

- For error checking:

if else
n = 0
while n == 0:
- s = input("Enter a number:")
- if s.lstrip('-').isdigit() == True:
-- n = int(s)
- elif s.lstrip('-').isdigit() == False:
-- print(s, " is not an integer.")
- else:
-- print("Something went wrong.")

exception handling
n = 0
while n == 0:
- try:
-- s = input("Enter a number:")
-- n = int(s)
- except ValueError as e:
-- print(s, "is not an integer.") - else:
-- print("Something went wrong.")

Raising Exceptions

- If you want the same thing to printed every time a specific error occurs, you can "raise" it at the beginning of the program to tell it what to print in the case of that error occuring
- Can only tell it what to print, cannot tell it to do anything else
- Later on if you can specified that if that error occurs in that section of code to do a specific thing, it will print what you raised earlier, and then do that thing

- In code:

if (invalid):
- raise ValueError("Invalid import.")

Exception Types

- Some common exception types:
  - TypeError: Indicates that a variable has the wrong *type* for an operation being used
    - Eg, Trying to use a float for an interger function
  - ValueError: Indicates the variable is the right *type* but is the wrong *value*
    - Eg, Trying to use a number less than 0, when it needs to be greater than 0
  - NotImplementedError: When a classes method need to be implemented in a child class