

Review

COMP3522 OBJECT ORIENTED PROGRAMMING 2

WEEK 6

Pass by assignment

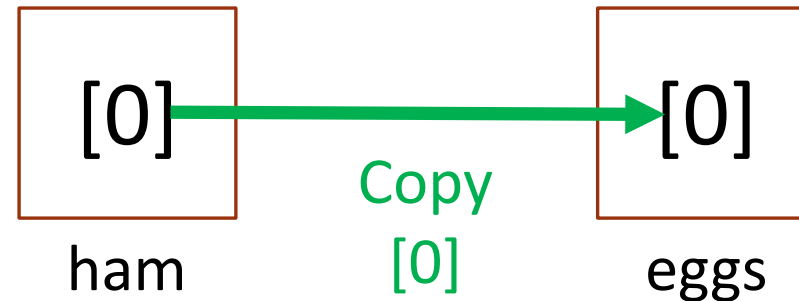
Pass by value

Pass by value is when parameters in a function are a COPY of the variables passed in

Any changes to the parameter do NOT change the value of the original variable passed in

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```



eggs is a new parameter variable. A new piece of memory is created and a copy of [0] is placed into the memory of eggs. Any changes to egg do NOT affect ham because eggs has its own place in memory

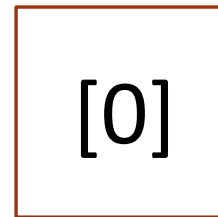
Pass by value

Pass by value is when parameters in a function are a COPY of the variables passed in

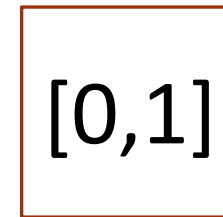
Any changes to the parameter do NOT change the value of the original variable passed in

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```



ham



eggs

eggs is a new parameter variable. A new piece of memory is created and a copy of [0] is placed into the memory of eggs. Any changes to egg do NOT affect ham because eggs has its own place in memory

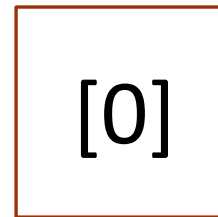
Pass by value

Pass by value is when parameters in a function are a COPY of the variables passed in

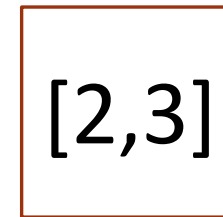
Any changes to the parameter do NOT change the value of the original variable passed in

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```



ham



eggs

eggs is a new parameter variable. A new piece of memory is created and a copy of [0] is placed into the memory of eggs. Any changes to egg do NOT affect ham because eggs has its own place in memory

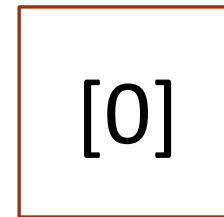
Pass by reference

Pass by reference is when parameters in a function are referencing the variables passed in

Any changes to the parameter CHANGE the value of the original variable passed in

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```



ham
eggs

eggs is referencing the same memory as ham. Any changes to eggs directly change ham

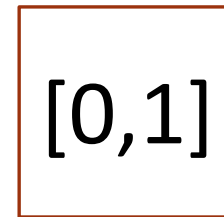
Pass by reference

Pass by reference is when parameters in a function are referencing the variables passed in

Any changes to the parameter CHANGE the value of the original variable passed in

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```



[0,1]

ham
eggs

eggs is referencing the same memory as ham. Any changes to eggs directly change ham

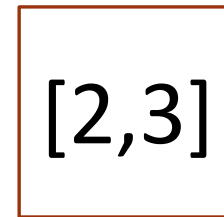
Pass by reference

Pass by reference is when parameters in a function are referencing the variables passed in

Any changes to the parameter CHANGE the value of the original variable passed in

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```



[2,3]

ham
eggs

eggs is referencing the same memory as ham. Any changes to eggs directly change ham

Pass by assignment - Mutable vs immutable parameters

Python handles memory differently. Let's call it **pass by assignment**

Depending on if the parameter is mutable or immutable different behaviors occur

Mutable parameters

- Can make functions appear to be pass by reference
- Mutable objects can change the internal attributes
- Changes will affect the original object
- Assignment causes parameter to point to a different object. Causes function to appear to be pass by value

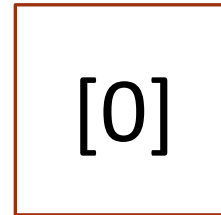
Immutable parameters

- Can make functions appear to be pass by value
- Immutable objects do NOT change the internal attributes. This is consistent with the concept of immutable types
- Any changes that appear to occur are actually new immutable objects that are created

Mutable function parameter

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```



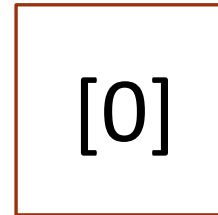
ham

A new object is created in memory, and the variable `ham` is assigned to it

Mutable function parameter

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```



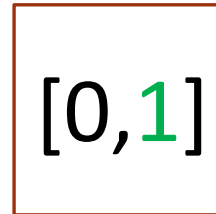
ham
eggs

eggs is a new variable that points to the same value ham is pointing at. Eggs and ham are both pointing at the same object in memory. There is NO COPYING

Mutable function parameter

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```



[0, 1]

ham
eggs

eggs has the value 1 appended to it. This changes the original value ham is assigned to

Mutable function parameter

```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```

[0,1]

ham

[2, 3]

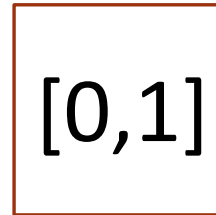
eggs

The object [2,3] is created in memory and eggs has now changed to point to it

Mutable function parameter

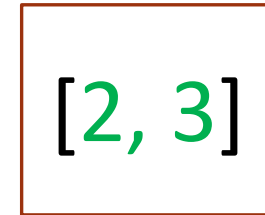
```
def spam(eggs):  
    eggs.append(1)  
    eggs = [2, 3]
```

```
ham = [0]  
spam(ham)  
print(ham)
```



[0,1]

ham



[2, 3]

eggs

When we print ham, its value is [0,1]

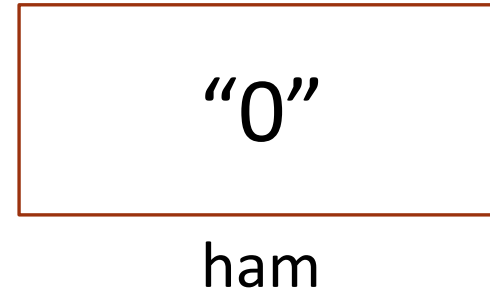
Mutable parameters appear to treat function parameters as pass by reference. Changes made internally to the mutable parameter change the original object

However assignment (=) causes the parameter to point to different objects

Immutable function parameter

```
def spam(eggs):  
    eggs += "1"  
    eggs = "2,3"
```

```
ham = "0"  
spam(ham)  
print(ham)
```

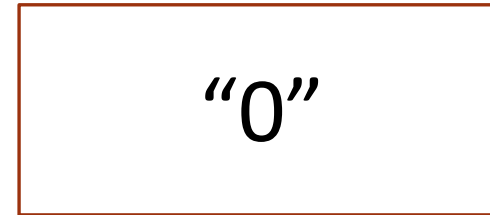


A new immutable string object is created in memory, and the variable `ham` is assigned to it

Immutable function parameter

```
def spam(eggs):  
    eggs += "1"  
    eggs = "2,3"
```

```
ham = "0"  
spam(ham)  
print(ham)
```

A rectangular box with a thin red border, containing the text "0" in a large, black, sans-serif font. This represents a memory object for the string "0".

ham
eggs

"0" is passed as a parameter to eggs. Before executing the first line, eggs and ham are both pointing at the same "0" object

Immutable function parameter

```
def spam(eggs):  
    eggs += "1"  
    eggs = "2,3"
```

```
ham = "0"  
spam(ham)  
print(ham)
```

"0"

ham
eggs

"01"

Strings are immutable. Meaning that we can't change a string in memory that already exists
To append "1" to "0" requires creating an entirely new string

Immutable function parameter

```
def spam(eggs):  
    eggs += "1"  
    eggs = "2,3"
```

```
ham = "0"  
spam(ham)  
print(ham)
```

"0"

ham

"01"

eggs

When we assign "01" to eggs, this makes egg point to the new "01" object

Immutable function parameter

"2,3"

```
def spam(eggs):  
    eggs += "1"  
    eggs = "2,3"
```

```
ham = "0"  
spam(ham)  
print(ham)
```

"0"

ham

"01"

eggs

A new string object is created on the right side of the assignment statement

Immutable function parameter

```
def spam(eggs):  
    eggs += "1"  
    eggs = "2,3"
```

```
ham = "0"  
spam(ham)  
print(ham)
```

"2,3"

eggs

"0"

ham

"01"

Eggs is now pointing to the new "2,3" string

Immutable function parameter

```
def spam(eggs):  
    eggs += "1"  
    eggs = "2,3"
```

```
ham = "0"  
spam(ham)  
print(ham)
```

"0"

ham

"01"

eggs

ham is still pointing at the string "0" so it prints out "0"

Immutable parameters make the function appear to treat parameters as pass by value

Method Resolution Order

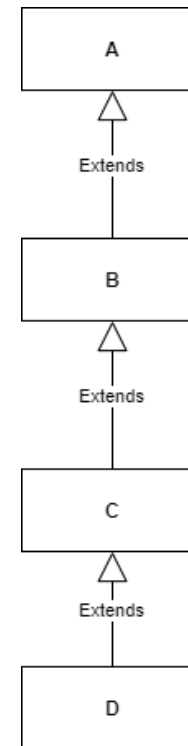
Method Resolution Order

Method Resolution Order (MRO) is the **order** that Python looks for a **method** in a hierarchy of classes

It's trivial when dealing with single inheritance

Begin from the class you want to determine the MRO

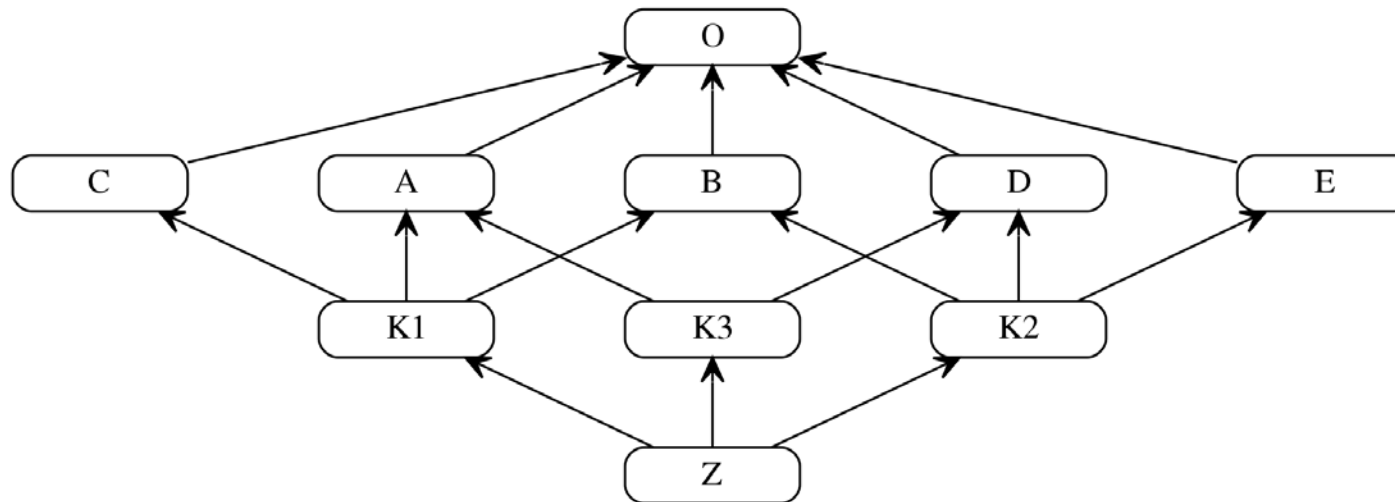
- Let's find the MRO of D
- And follow up the inheritance tree to the base class A
- $L(D) = [D, C, B, A]$



Method Resolution Order

However things get a lot trickier when we need to find the MRO with multiple inheritance

We have to remember two main things, MRO must respect **local precedence ordering**, and also provide **monotonicity**



MRO: Rules

Local Precedence ordering:

The order in a class that we're inheriting from multiple parents

This means that a class $C(B1, \dots, Bn)$ will have in its MRO, the base classes $(B1, \dots, Bn)$ in the order that we specified

```
class F:
    remember2buy = "Milk"
class E(F):
    remember2buy = "Eggs"
class G(E, F):
    pass
```

Local precedence of:

- class F is the implicit (**object**)
- class E is (**F**)
- class G is (**E, F**)

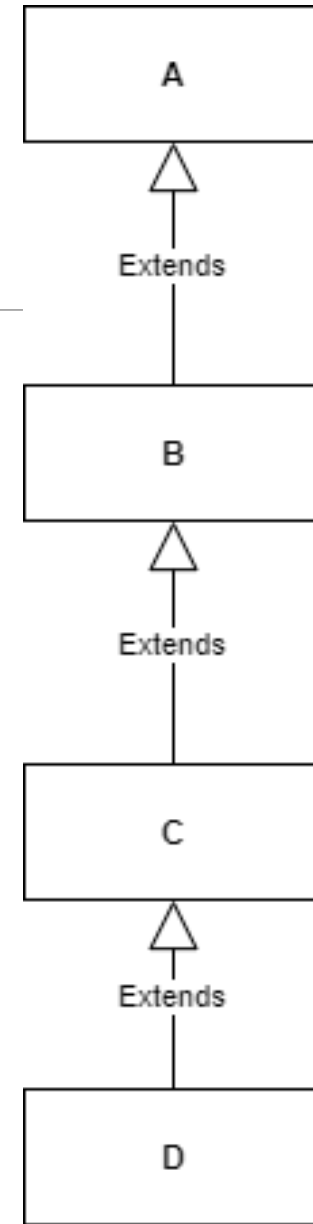
MRO: Rules

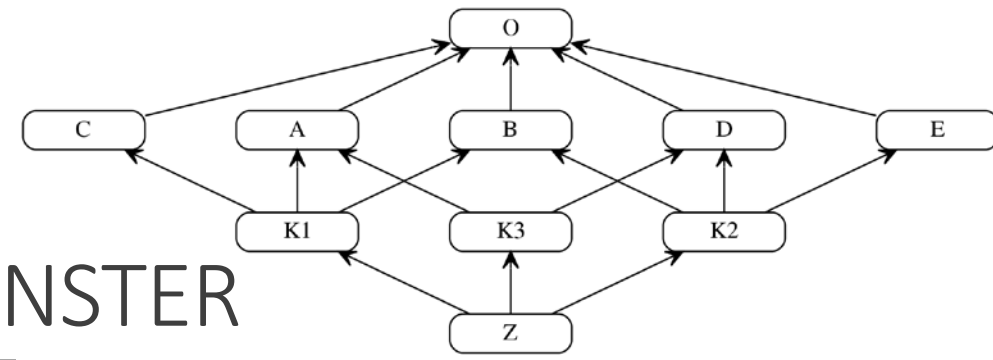
Monotonicity:

If B Precedes A in the Linearization of C, then B **HAS TO** precede A in the Linearization of any child class of C.

Linearization(C) = [C, **B**, **A**, object]

Linearization(D) = [D, C, **B**, **A**, object]





Try finding out the MRO of this MONSTER

$L(O) = [O]$

$L(A) = [A, O]$

$L(B) = [B, O]$

$L(C) = [C, O]$

$L(D) = [D, O]$

$L(E) = [E, O]$

$L(K1) = [K1, A, B, C, O]$

$L(K2) = [K2, D, B, E, O]$

$L(K3) = [K3, D, A, O]$

class Z extends K1, K2, K3 #original python class code

$L(Z) = [Z] + \text{merge}(L(K1), L(K2), L(K3), [K1, K2, K3])$

What this is saying in regular English is “To find the **linearization of Z**, we must start with the **list Z** and merge it with the **linearization of K1, K2, K3**, along with the immediate parents **K1, K2, K3**”

Let's try it out together in a word doc!

Customer Iterable

Custom iterable

An iterable is an object that will return an iterator when it's call with `iter(custom_object)`

All that needs to be done in the custom iterable is to override the `__iter__()` method and return an appropriate iterator

The returned iterator can be an iterator from an existing iterable type like a list, tuple, dictionary or your own custom iterator

Custom iterable

```
class custom_team:
    def __init__(self, team_name):
        self.team_name = team_name
        self.list = []

    def append_member(self, m):
        self.list.append(m)

    def __iter__(self):
        return iter(self.list)
```

```
ct = custom_team("Simpsons")
ct.append_member('Marge')
ct.append_member('Homer')

my_iter = iter(ct)
print(next(my_iter))
print(next(my_iter))
```

Class custom_team contains a list

By default custom_team is not iterable

By adding `__iter__` the custom_team becomes iterable, but it's iterable because we're instantiating an iterator from self.list

Custom iterable

Chances are when you're storing data in a custom class, you'll use an existing container type (list, tuple, dictionary etc)

So it's trivial to create and return an iterator based on existing container types

Things get trickier when creating your own custom containers not based off existing containers

- Imagine creating a linked list from scratch
- You will need to create a custom iterator that can iterate through the linked list
- Any custom class that uses the linked list and wants to be iterable would then instantiate that custom iterator in the `__iter__` function and return it

[linked_list_iterator.py](#)

Keyword arguments - Kwargs

Variable Keyword Arguments

Named Arguments are also known as Keyword Arguments.

Just like we can accept an arbitrary variable number of arguments using `*args`

`*args` is treated like a sequence type

We can accept an arbitrary variable number of **keyword arguments** using `**kwargs`

`**kwargs` is treated like a dictionary

Variable Keyword Arguments

```
def variable_keyword_args(**kwargs):  
    for key, item in kwargs.items():  
        print(key, item)
```

```
variable_keyword_args(port=21, host='localhost', debug=True)
```

There are keys

A diagram consisting of three orange arrows. All three arrows originate from a single point below the text 'There are keys'. One arrow points to the word 'port' in the function call, another points to the word 'host', and the third points to the word 'debug'. The words 'port', 'host', and 'debug' are highlighted in red in the original image.

Variable Keyword Arguments

```
def variable_keyword_args(**kwargs):  
    for key, item in kwargs.items():  
        print(key, item)
```

```
variable_keyword_args(port=21, host='localhost', debug=True)
```



There are keys, and values

Variable Keyword Arguments

```
def variable_keyword_args(**kwargs):  
    for key, item in kwargs.items():  
        print(key, item)
```

```
variable_keyword_args(port=21, host='localhost', debug=True)
```

****kwargs** interprets the key/value parameters passed in as a dictionary

Variable Keyword Arguments

```
def variable_keyword_args(**kwargs):  
    for key, item in kwargs.items():  
        print(key, item)
```

```
variable_keyword_args(port=21, host='localhost', debug=True)
```

kwargs is a dictionary, so in order to get the key/value pairs, we need to explicitly call the **items method**

Notice how we're using *unpacking* to get the key/value pairs from the dictionary

[kwargs_example.py](#)

SOLID Design Principles

- A set of guidelines to help us write Maintainable, decoupled, flexible Object Oriented Code
- Created by Robert C. Martin, also known as Uncle Bob amongst developers.
- Highly respected and has a bunch of books on writing clean code.



Single Responsibility Principle

A class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class)



Open / Closed Principle

A software module (it can be a class or method) should be open for extension but closed for modification.



Liskov Substitution Principle

Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.



Interface Segregation Principle

Clients should not be forced to depend upon the interfaces that they do not use.



Dependency Inversion Principle

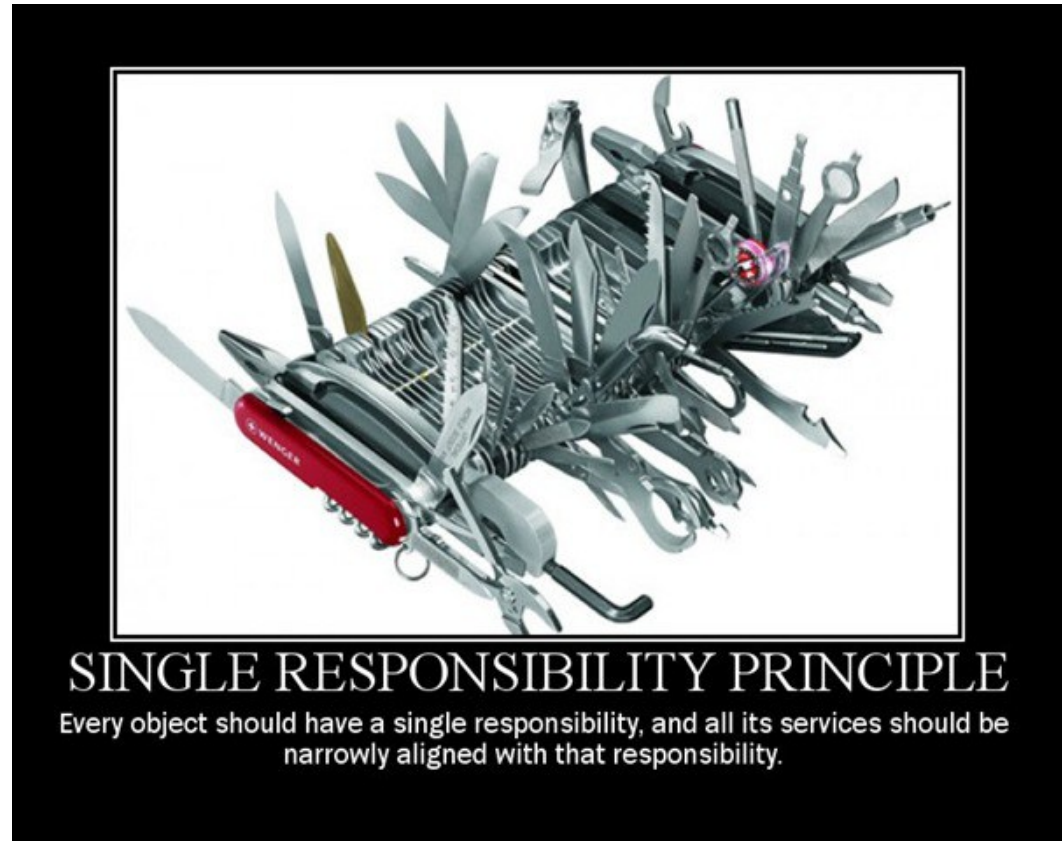
Program to an interface, not to an implementation.

Single Responsibility Principle

“The Single Responsibility Principle requires that each class is responsible for only one thing.”

Ask yourself, are there multiple reasons to change my class?

There should only be one reason to change your class. Each class should have one and only one responsibility.



Single Responsibility Principle

Think of the Transaction class in Assignment 1

Conceptually what is a transaction in the real world?

In physical form it's kind of like a receipt. It indicates

- Money amount
- Location
- Date/time

In the real world, does the receipt also have the ability to

- Query you for information?
- Ask you about your budget?
- Have access to your bank account?

Single Responsibility Principle

The **single responsibility** of the transaction class is to contain information about a transaction (money, location, date/time etc)

It should not have any other responsibility (UI interactions, access to other classes)

Other responsibilities should be handled by other classes

- To create a new transaction object:
- UI Manager's single responsibility is to query user about different types of information and return that information to whoever needs it
- User class' single responsibility is to contain all the information that makes up a user
- If one class becomes too bloated and is taking on too much responsibility, split it up into multiple classes

Open Closed Principle

The Open/Closed Principle states that software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

Once a class is made, it's behaviour shouldn't be modified to support special cases. It should be **CLOSED for modification**.

Instead, we extend (or inherit) from the class and override behaviours. That is, it's **OPEN for extension**.



How do we do this?

We accomplish this by:

1. Partitioning our software system into components
2. Arranging the components into a dependency hierarchy that protects higher-level components from changes in lower level components
3. Avoiding modifying source code in classes that are already in use
4. Using abstract base classes in inheritance hierarchies as interfaces

Our goal is (as always) to make a system easy to extend without incurring a high impact of change

Software changes

We don't want to modify code that's in use

But sometimes we have to make changes

We **should extend the original object:**

- Leave original source code untouched.

Sounds like polymorphism doesn't it!

Challenges:

- We have to anticipate changes
- Designer must be aware of possible variations in current functionality, consider a technical interface that will scale.

Example

Imagine we're asked to write code to calculate area of a rectangle

```
class Rectangle:
    def __init__(self, w, h):
        self.width = w
        self.height = h

def area(r):
    return r.width*r.height
```

Example

Asked to now find area of rectangle and circle

```
class Shape:
    pass

class Rectangle(Shape):
    def __init__(self, w, h):
        self.width = w
        self.height = h

class Circle(Shape):
    def __init__(self, r):
        self.radius = r

def area(shape):
    if shape is Rectangle:
        return shape.width*shape.height
    else:
        return shape.radius * shape.radius * 3.14
```

Example

Asked to now find area of rectangle and circle, and triangle...

```
class Shape:
    pass

class Rectangle(Shape):
    def __init__(self, w, h):
        self.width = w
        self.height = h

class Circle(Shape):
    def __init__(self, r):
        self.radius = r

class Triangle(Shape):
    #triangle code
    pass

def area(shape):
    if shape is Rectangle:
        return shape.width*shape.height
    elif shape is Circle:
        return shape.radius * shape.radius * 3.14
    else:
        #triangle area code
        pass
```

Example - Solution

Abstract out shape and have subtypes have their own area functions

After creating the shape abstract class and submitting it to repo, it's considered **closed** for modification. But it's always **open** to extend functionality through subclasses

```
class Shape(abc.ABC):  
    @abc.abstractmethod  
    def area(self):  
        pass
```

```
class Rectangle(Shape):  
    def __init__(self, w, h):  
        self.width = w  
        self.height = h  
    def area(self):  
        return self.width * self.height
```

```
class Circle(Shape):  
    def __init__(self, r):  
        self.radius = r  
    def area(self):  
        return self.radius * self.radius * 3.14
```

```
def area(shape):  
    return shape.area()
```


Liskov Substitution Principle

In programming, the Liskov substitution principle states that if S is a subtype of T , then objects of type T may be replaced (or substituted) with objects of type S .

Or

Objects in a program should be replaceable with instances of their base types without altering the correctness of that program.



Liskov Substitution Principle

We can **substitute** passing in a Shape, Rectangle, or Circle object into the **area** function, and it should work

If for some reason calling shape.area() doesn't work with Shape or any of its subclasses, then the function fails the liskov substitution principle

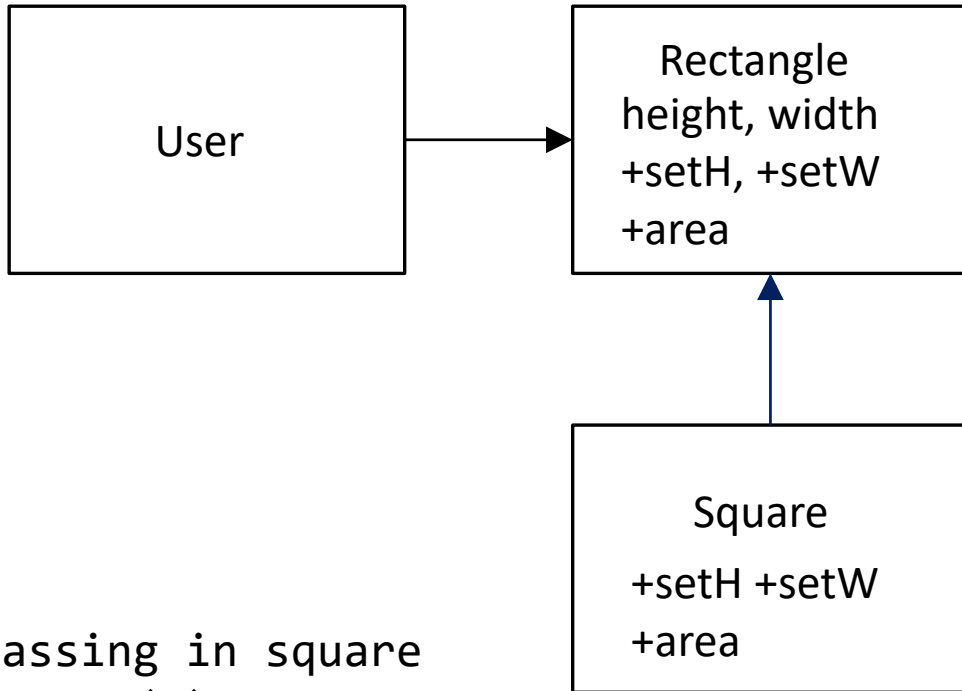
```
class Shape(abc.ABC):  
    @abc.abstractmethod  
    def area(self):  
        return -1
```

```
class Rectangle(Shape):  
    def __init__(self, w, h):  
        self.width = w  
        self.height = h  
    def area(self):  
        return self.width * self.height
```

```
class Circle(Shape):  
    def __init__(self, r):  
        self.radius = r  
    def area(self):  
        return self.radius * self.radius * 3.14
```

```
def area(shape : Shape):  
    return shape.area()
```

Failing Liskov Substitution Principle



```
//passing in square
def area(r)
  r.setW(5); //sets height & width 5
  r.setH(2); //sets height & width 2
  assert(r.area() == 10); // Error area = 4
```

Fails Liskov Substitution Principle

Square is not a proper subtype of Rectangle

The height and width of rectangle are independently mutable

In contrast, the height and width of square must change together

The user will get confused because they will think they are dealing with a rectangle

Interface Segregation Principle

Many client-specific interfaces are better than one general-purpose interface.

Or

The interface segregation principle states that no client should be forced to depend on methods it does not use.

Put more simply: Do not add additional functionality to an existing interface by adding new methods.

This is the Single Responsibility Principle for Interfaces/Abstractions!



Back To Our Zoo!

```
class Walkable(abc.ABC):
```

```
    @abc.abstractmethod
    def walk(self):
        pass
```

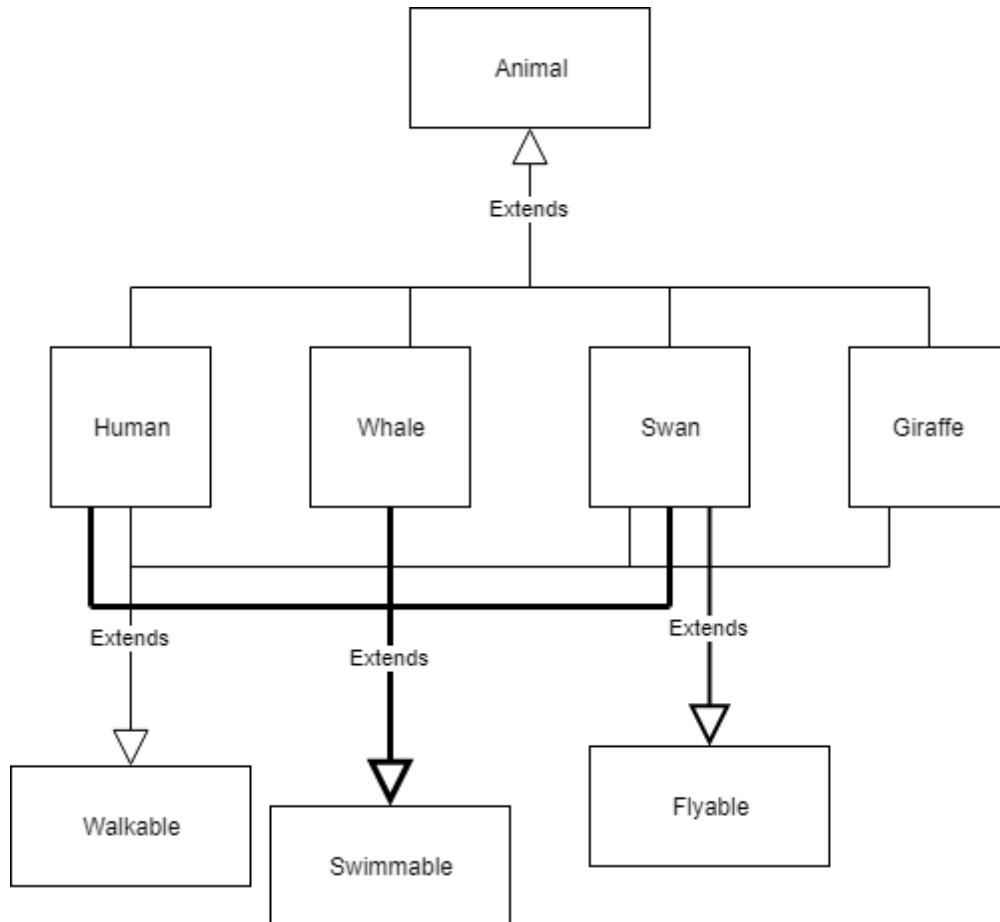
```
class Swimmable(abc.ABC):
```

```
    @abc.abstractmethod
    def swim(self):
        pass
```

```
class Human(Animal, Walkable, Swimmable):
```

```
    def walk(self):
        # overridden walking code
```

```
    def swim(self):
        # overridden swimming code
```



Instead of polluting the animal base class with additional functionality, we create separate interfaces (read: ABC) that handle different responsibilities.

This may seem difficult to maintain, but in fact it isn't.

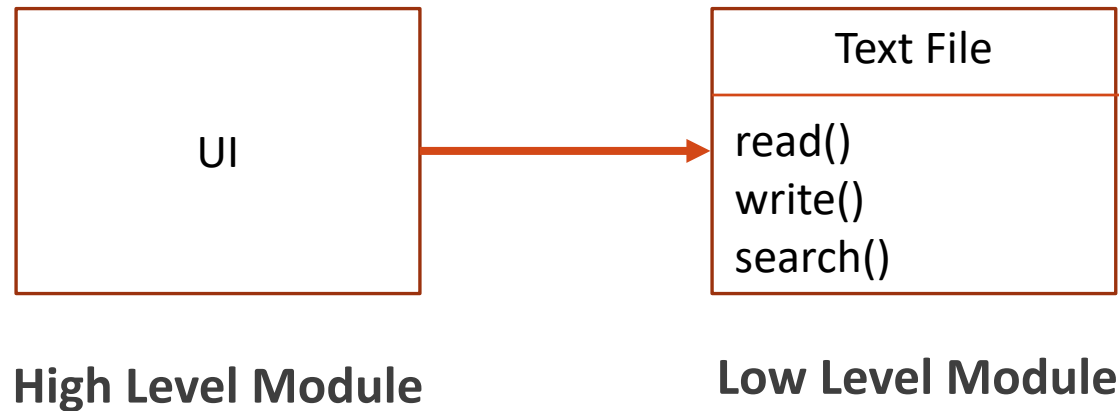
Dependency Inversion Principle

Dependency Inversion Principle can be thought as a combined effect of the previous principles.

Specifically, the Open Closed Principle and Liskov Substitution Principle

Before we dive into this, we need to look at a few concepts.

High Level & Low Level Modules

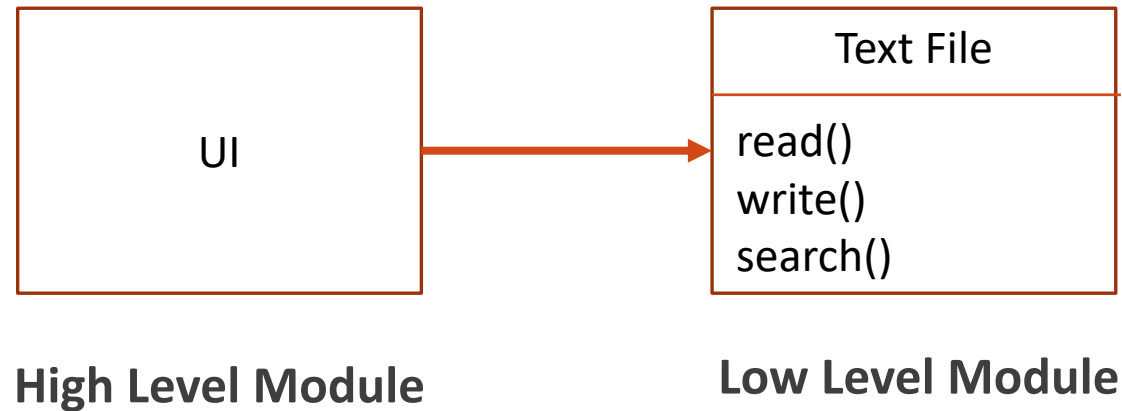


High Level Modules are the ones that contain complex logic made up of small ‘chunks’ of simpler code. For example, our UI class

A **Low Level Module** on the other hand, is a class that encapsulates some simple atomic behaviour. Such as writing and reading from a file, displaying an image, etc.

We generally compose high level modules with low level modules. That is, **high level modules are Coupled and Dependent on low level modules**

High Level & Low Level Modules



This is not very good design.

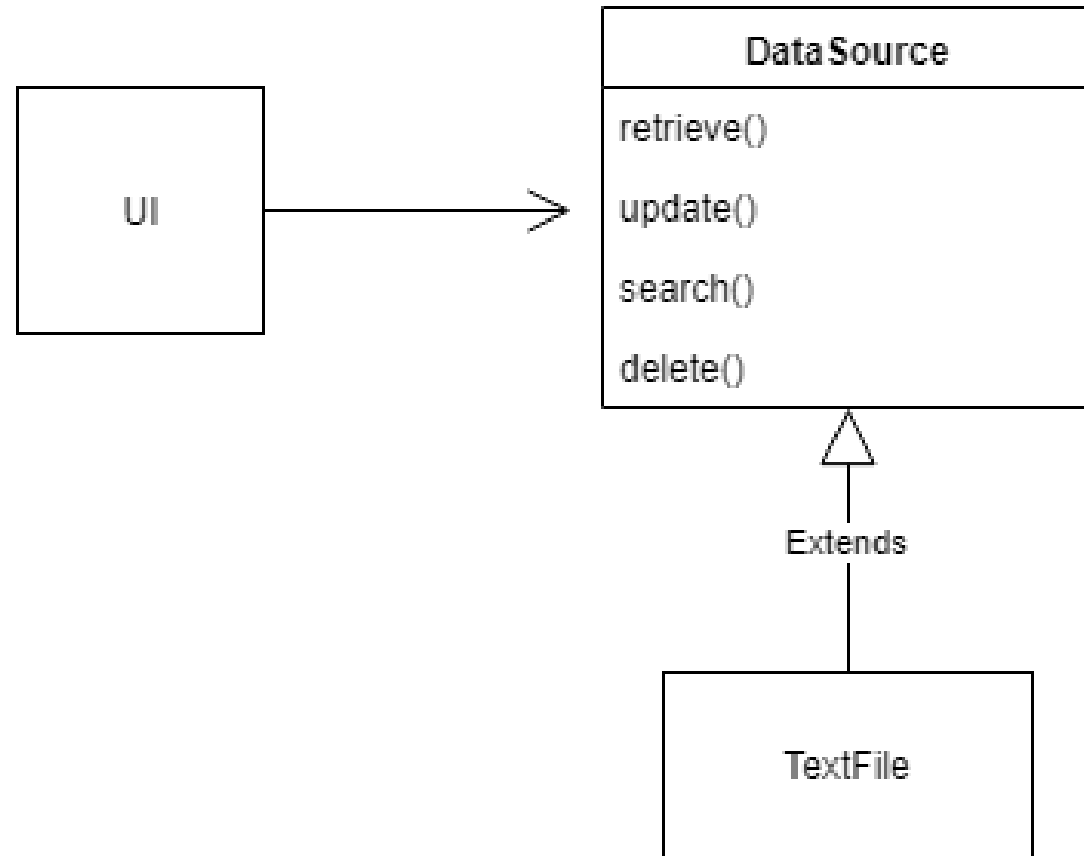
We don't want this dependency. We want to be able to change or replace the lower level modules without tampering with the complex code in the higher level modules

We inversed the dependencies!

We introduced an abstract layer, a middle man that does not depend on anything.

We made both our high level and low level modules depend on the abstract instead.

Now we can switch out either side, the UI and the text file could be replaced with something completely different



Dependency Inversion Principle

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

We call classes that implement details 'Concrete Classes'. In the example we just saw, the Data Source is our abstract class or Interface, and the UI and TextFile classes are concrete classes.

Abstract classes declare an interface.

Concrete classes actually contain the code that implements the details of the interface.

Law of Demeter

“Each unit should have only limited knowledge about other units: only units “closely” related to the current unit. Each unit should only talk to its friends; don’t talk to strangers.”

Don’t have train wrecks like these:

```
obj.getX().getY().getZ().doSomething();
```

Or

```
MyCharacterUI.GetCharacterData().CreateCharacter().Save()
```

Poor Demeter is often forgotten, some people even call it “Suggestion of Demeter” since this is difficult to avoid

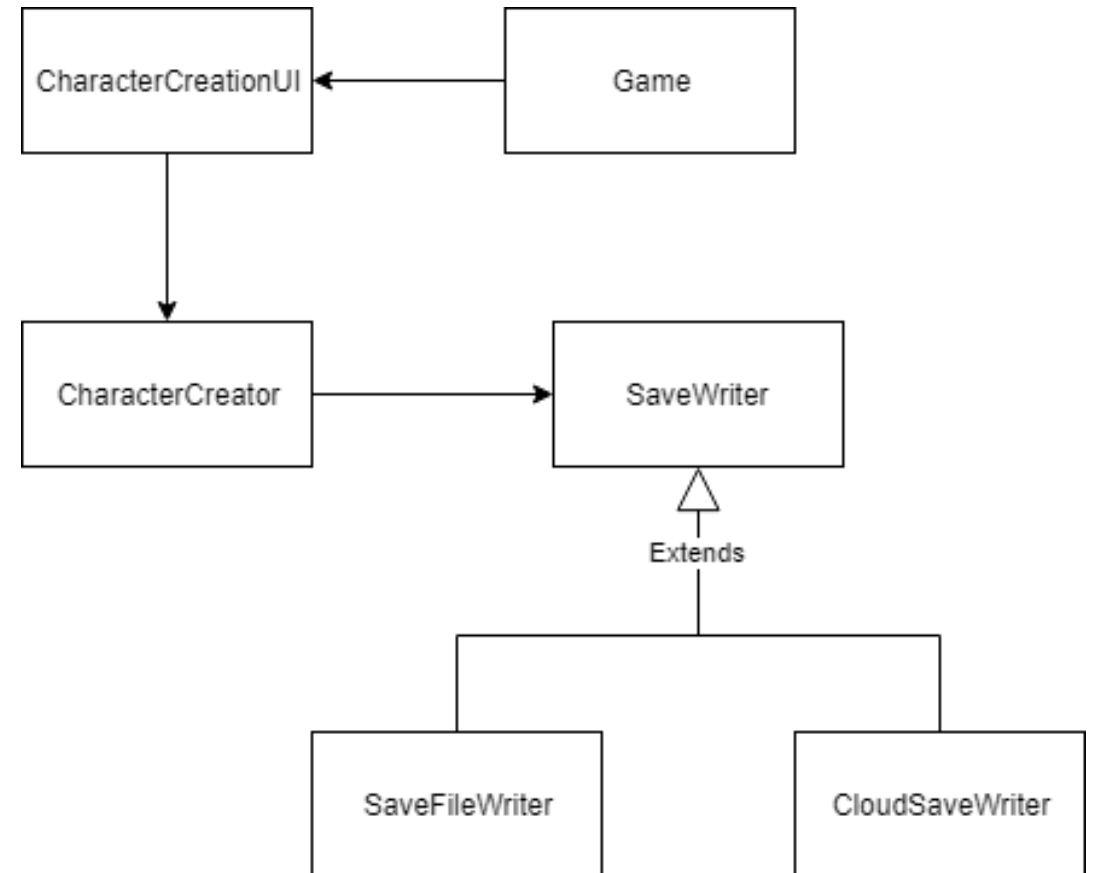
Law of Demeter – Example Code

```
class game:
    def create_character(self):
        data = self.character_ui.get_character_creator().
            save(self.character_ui
                .get_character_creator()
                .create_character()
            )
```

This is obviously bad and hard to read code

I also had a hard time getting the lines to wrap in powerpoint. Bad for many reasons

Any change in the save method, which is all the way in SaveWriter, could have a ripple effect of changes all the way to our game class

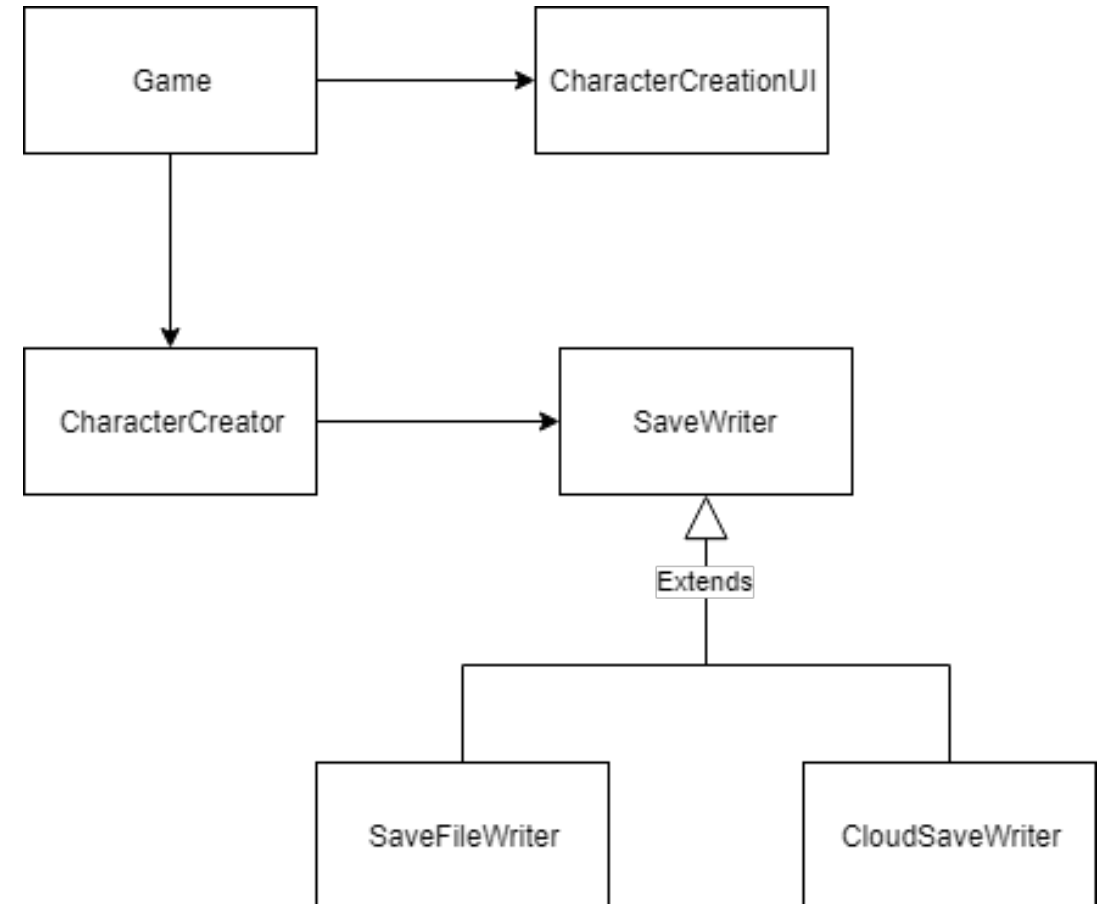


Law of Demeter – Example Code - Solution

```
class game:
    def create_character(self):
        data = self.character_ui.get_data()
        self.character_creator
            .initiate_character_creation(data)

class CharacterCreationUI:
    def get_data(self):
        return self.character_data

class CharacterCreator:
    def initiate_character_creation(self, data):
        my_char = self.create_character(data)
        self.save_manager.save(my_char)
```



That's it for this week!

Best of Luck with the Midterms

All of you are awesome and have really impressed me with your progress.

Focus on the materials, not the marks.

Go over the slides alongside the sample code

Go over the past quizzes

