OOP Revision Notes

Motivation: Why do we need Object Oriented Programming?

- Programs and **Applications** often **solve** or **interact** with **real-world objects**
  - A code is written to **achieve some goal** in the real world
- OOP helps to **represent real-world scenarios**
- To **simulate or replicate real-world objects** in our code
- **Example** - A **Banking Application** has:
  - **Customers**
  - **Bank Accounts**
  - **Transactions**
- Even in our code, we will have some **representations of these objects** existing in real-world
  - Representation of Customers, Accounts,
  - Representations of Transactions happening among these Accounts

Example

Let's say:

- We have a **School**
- It has **Students** in it

OOP will allow us to:

- **Write the code once** for a given object like Student
- **Reuse that code** as many times as we want the objects (Students)
- Write **template code** for objects having similar **behaviours/characteristics/properties**

- Use that code to create as many similar objects as we want

To implement OOP, we need to know 2 things:

1. **Class**
    - It's a **Template**
    - A **Blue Print**
    - Use it to **create as many objects** as we want
    - **Example**: `class Student`
2. **Object**
    - **Instances** of a class
    - **Created using the blueprint code of class**
    - **Example**: `Anant` is an instance of `Student`

A class is created using the keyword `class`

- The **convention** is to **start the class name** with **uppercase letter**

A class contains:

1. **Properties** - **Data** or Variables which have some values
2. **Methods** - Functions which are used to perform some **mutations** or tasks on object's properties

So, **data and its mutations are stored together inside a class**

Let's create our first `class` and its object

```
class Student:
    pass


s = Student()


s


<__main__.Student at 0x7fd1f8f78880>


type(s)
```

```
__main__.Student


s.name = "Rahul"


s.name


'Rahul'
```

Lets pre-define some characteristics

```
class Student: def __init__(self): self.name = "some name"
```

`__init__` is **short for inititlisation**

```
class Student:
    def __init__(self):
        self.name = None
```

Lets again try to initialise another object using updated class

```
s1 = Student()
print(s1.name)


None
```

Lets try to understand “self” a bit more by doing printing self

```
class Student:
    def __init__(self):
        print(self)
        self.name = "some name"


s1 = Student()
print(s1)


<__main__.Student object at 0x00000185CAFFFE80>
<__main__.Student object at 0x00000185CAFFFE80>
```

We can change the name property of our `Student` object:

```
s1.name = "Mudit"
print(s1.name)
```

```
Mudit
```

`__str__` returns the string representation of the object instead of its address

```
class Student:
    def __init__(self):
        self.name = "Anant"

    def __str__(self):
        return f"Student is {self.name}"


s1 = Student()
print(s1)
```

```
Student is Anant
```

```
s1 = Student()
s2 = Student()
s3 = Student()
```

Question: But there is a problem here. What is it?

All the students are going to get initialized with the same name `Anant`

```
print(s1.name)
print(s2.name)
print(s3.name)
```

```
Anant
Anant
Anant
```

Well, like we saw before

- We can **change the property values of `Student` objects**

```
s2.name = "Mudit"
s3.name = "Priya"

print(s1.name)
print(s2.name)
print(s3.name)
```

```
Anant
Mudit
Priya
```

But wee don't want to start with same default name for every student

`__init__()` function allows us to pass other parameters

We can **use parameter values passed to `__init__()` to initialize properties**

```
class Student:
    def __init__(self, newName):
        self.name = newName

    def __str__(self):
        return f"Student is {self.name}"


s1 = Student("Anant")
s2 = Student("Mudit")
s3 = Student("Priya")

print(s1.name)
print(s2.name)
print(s3.name)
```

```
Anant
Mudit
Priya
```

Now what if we create a new `Student` without passing any name as argument?

```
s2 = Student()
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-18-e2d9b8a0dd10> in <module>
----> 1 s2 = Student()


TypeError: __init__() missing 1 required positional argument: 'newName'
```

- We get an **Error!!**
- `__init__` function **got less arguments than it was supposed to**

Takeaway from this:

- Whatever **parameters we define for `__init__` function** in our class definition

- We need to **pass that many arguments** while creating an object of that class

Adding more Attributes to the class blueprint

Let's say we want to assign a **Roll Number** to each student that we are creating

```
class Student:
    def __init__(self, newName):
        self.name = newName
        self.rollNum = 0

    def __str__(self):
        return f"Student is {self.name}"
```

Just like last time,

- **Every new student object created will have a Roll Number of 0**
- We will have to **hard code or manually change** everyone's roll number

```
class Student:
    def __init__(self, newName, rollNum):
        self.name = newName
        self.rollNum = rollNum

    def __str__(self):
        return f"Student is {self.name}"

s1 = Student("Anant", 101)
```

Let's also **print** the **name** and **Roll No.** of `s1`

```
print(s1.name)
print(s1.rollNum)

# Let's also print string representation of s1

print(s1)
```

```
Anant
101
Student is Anant
```

```
class Student:
    def __init__(self, newName, newRollNum):
        self.name = newName
        self.rollNum = newRollNum

    def __str__(self):
        return f"{self.rollNum}. {self.name}"

s1 = Student("Anant", 101)
s2 = Student("Mudit", 102)
s3 = Student("Priya", 103)

print(s1)
print(s2)
print(s3)
```

```
101. Anant
102. Mudit
103. Priya
```

Notice how convenient it is

- As soon as we **make changes in the blueprint class**
- **All its objects start reflecting that change**
- This is a **huge benefit of OOP paradigm**
    - We just need to **write the blueprint code once**
    - We can use that code as many times as we want
    - **All objects created using that blueprint will follow those properties**

Now, Remember how we got error?

when we did not pass a `rollNum` value as argument while creating a `Student` object?

We can also pass default values of parameters inside `__init__` to avoid that error:

```
class Student:
    def __init__(self, newName, newRollNum=-1): # default value of rollNum is -
        self.name = newName
        self.rollNum = newRollNum

    def __str__(self):
        return f"{self.rollNum}. {self.name}"

s1 = Student("Anant")
s2 = Student("Mudit")
s3 = Student("Priya", 103)

print(s1)
print(s2)
print(s3)
```

```
-1. Anant
-1. Mudit
103. Priya
```

Makee sure parameters with default values are in the end !!

```
class Student:
    def __init__(self, newName="NO_NAME", newRollNum):
        self.name = newName
        self.rollNum = newRollNum
```

```
  File "<ipython-input-6-69f809b3bea9>", line 2
    def __init__(self, newName="NO_NAME", newRollNum):
                 ^
SyntaxError: non-default argument follows default argument
```

All parameters with default values should be AFTER parameters with no default values

```
class Student:
    def __init__(self, newRollNum, newName="NO_NAME"):
        self.name = newName
        self.rollNum = newRollNum
```

What does `self` mean?

But what does this `self` mean?

- First of all, `self` is not a **keyword**
- It is a **reference to an object**
- It **tells the class which of its objects we are referring to**

When we print `s1.name` :

```
print(s1.name)
```

```
Anant
```

- It prints out the **value of `self.name` for object `s1`**
- So, the class knows it needs to **give value of `name` for its object `s1`**
- **Every object has a different value of its attributes**
    - `s1` has `name = "Anant"` and `rollNum = 101`
    - `s2` has `name = "Mudit"` and `rollNum = 102`
    - `s3` has `name = "Priya"` and `rollNum = 103`
    - ... and so on
- `self` **helps the class to differentiate b/w its objects and values of their attributes**
- In a way, it **keeps a separate copy of attribute values for each object**

Automatically Assigning an Attribute Value

```
class Student:
    def __init__(self, newName): # counter inside `__init__`
        self.name = newName
        self.counter = 0 # initialize counter to 0
        self.counter += 1 # increment counter by 1
        self.rollNum = self.counter # assign counter to rollNum

    def __str__(self):
        return f"{self.rollNum}. {self.name}"

s1 = Student("Anant")
s2 = Student("Mudit")
s3 = Student("Priya")

print(s1)
print(s2)
print(s3)
```

```
1. Anant
1. Mudit
1. Priya
```

Clearly, it DID NOT work

- **Every student** got assigned **`rollNum = 1`**

How can we get a combined single counter for all the objects?

So that we can **track the number of objects** that are being created

- `counter` is **not to be assoicated** with every object
  - It is **not a property/attribute of every object**
- `counter` is **to be assoicated** with the class
  - **class is common to all the objects**

```
class Student:

    counter = 0 # initialize

    def __init__(self, newName):
        self.name = newName
        Student.counter += 1 # increment counter when new object is created
        self.rollNum = Student.counter # assign roll number to counter

    def __str__(self):
        return f"{self.rollNum}. {self.name}"

s1 = Student("Anant")
s2 = Student("Mudit")
s3 = Student("Priya")

print(s1)
print(s2)
print(s3)
```

```
1. Anant
2. Mudit
3. Priya
```

Now it's working :)

```
class Student:

    counter = 100 # initialize

    def __init__(self, newName):
        self.name = newName
        Student.counter += 1 # increment counter when new object is created
        self.rollNum = Student.counter # assign roll number to counter

    def __str__(self):
        return f"{self.rollNum}. {self.name}"

s1 = Student("Anant")
s2 = Student("Mudit")
s3 = Student("Priya")

print(s1)
print(s2)
print(s3)
```

```
101. Anant
102. Mudit
103. Priya
```

This leads us to the concept of `Class Variables`

- Class Variables are **associated with the class**
- **Common to all objects created**
- Class Variables have a **single copy for all the objects that we create using that class**

Class variables are accessed using class_name

```
print(Student.counter)
```

```
103
```

What happens if we access class variable `counter` using objects?

```
print(s1.counter)
print(s2.counter)
print(s3.counter)
```

```
103
103
103
```

- It gives the same value for all the objects
- Proves that **class variable** is **common to all the class objects**

We can also change the value of class variables outside the class

```
Student.counter = 1000

print(Student.counter)

print(s1.counter)
print(s2.counter)
print(s3.counter)
```

```
1000
1000
1000
1000
```

Can we change the value of a `Class Variable` using object's name?

```
class Student:

    counter = 100

    def __init__(self, newName):
        self.name = newName
        Student.counter += 1
        self.rollNum = Student.counter

    def __str__(self):
        return f"{self.rollNum}. {self.name}"

s1 = Student("Anant")
s2 = Student("Mudit")
s3 = Student("Priya")

print(s1)
print(s2)
print(s3)
print("-"*50)

s1.counter = 1000  # Changing value of Class Variable using object `s1`

print(Student.counter)
print(s1.counter)
print(s2.counter)
print(s3.counter)
```

```
101. Anant
102. Mudit
103. Priya
--------------------------------------------------
103
1000
103
103
```

How did this happen?

- When we did **`s1.counter = 1000`**

- Python **created a new attribute** `counter` for object `s1`
- and Python **set its value to** `1000`

Conclusion from this:

- We **can access class variable using class_name**
- We **can access class variable using objects**

But, correct or preferred way is to access using class_name, not objects

- Because a **class variable is not associated with any object**
- If we use an **object** to **change value of class variable**, a **new attribute gets created** for that object

Adding more Behaviours to the class

- We can **add more behaviours/functions** to our blueprint class
- These are called **Custom Methods**
- These are **not special functions like dunders** `__init__` **or** `__str__`
- These are **something that we create** to perform certain tasks

```
class Student:

    counter = 0

    def __init__(self, newName):
        self.name = newName
        Student.counter += 1
        self.rollNum = Student.counter

    def intro(self):
        print(f"Hello! My name is {self.name}")

    def __str__(self):
        return f"{self.rollNum}. {self.name}"

s1 = Student("Anant")
s1.intro()

# Now this works perfectly fine


Hello! My name is Anant
```

Exercise Question

Now it's your time to write the code

- Create a **class** called **`Account`**, which **refers to a bank account**

- Create **attributes** that will be unique for each instance of `Account`

1. `id` —> this has to be **incremented** and **assigned automatically**
2. `bal` —> this will give balance amount for each account
   - `bal` needs to be **assigned a value as soon as an account is created**
   - As soon as **account is created**, it should have some **opening balance**

- Create **2 instances** of accounts, **`a1`** and **`a2`**
  - `a1` should have `id = 1` and `bal = 100`
  - `a2` should have `id = next id` and `bal = 0`

- Create a **string representation** for each account
  - When we print an account, like `print(a1)`
  - It should print out:

`Account {id} has Rs. {balance}.`

```
class Account:

    counter = 0

    def __init__(self, openingBal=0):
        # Ask Ques: Can we write `counter`? or we need to write `Account.counte
        Account.counter += 1
        self.id = Account.counter
        self.bal = openingBal

    def __str__(self):
        # Ask Ques: Is it going to print or return a string?
        return f"Account {self.id} has Rs. {self.bal}"


a1 = Account(100)
a2 = Account()

print(a1)
print(a2)
```

```
Account 1 has Rs. 100
Account 2 has Rs. 0
```

Adding more functionality to `Account` class

Question: How will I keep track of account balance after depositing some amount?

```
class Account:

    counter = 0

    def __init__(self, openingBal=0):
        Account.counter += 1
        self.id = Account.counter
        self.bal = openingBal

    # Ask Ques: What should be the parameters of deposit()?
    def deposit(self, amount):
        self.bal += amount

    def __str__(self):
        # Ask Ques: Is it going to print or return a string?
        return f"Account {self.id} has Rs. {self.bal}"


a1 = Account(100)
a2 = Account()

print(a1)
print(a2)
```

```
Account 1 has Rs. 100
Account 2 has Rs. 0
```

```
# Let's use our deposit() method

a1.deposit(50)
print(a1)
print(a2)
```

```
Account 1 has Rs. 150
Account 2 has Rs. 0
```

Notice:

- Balance of **Account 1** gets **updated**
- Balance of **Account 2** still **remains same**

- Because we **only access and change `bal` of `a1` using `self.bal`**

What should happen in `withdraw()` method?

```
class Account:

    counter = 0

    def __init__(self, openingBal=0):
        Account.counter += 1
        self.id = Account.counter
        self.bal = openingBal

    def deposit(self, amount):
        self.bal += amount

    def withdraw(self, amount):
        self.bal -= amount

    def __str__(self):
        return f"Account {self.id} has Rs. {self.bal}"


a1 = Account(100)
a2 = Account()

print(a1)
print(a2)
```

```
Account 1 has Rs. 100
Account 2 has Rs. 0
```

```
# Let's use our withdraw() method on Account 1

a1.withdraw(50)

print(a1)
print(a2)
```

```
Account 1 has Rs. 50
Account 2 has Rs. 0
```

```
# Now Let's use our withdraw() method on Account 2

a2.withdraw(50)

print(a1)
print(a2)
```

```
Account 1 has Rs. 50
Account 2 has Rs. -50
```

Did you see the issue here?

Can account balance be negative in a real-world scenario?

Mostly NO

Also, let's try one more thing with `deposit()`

```
a1.deposit(-50) # depositing a negative amount

print(a1)
print(a2)
```

```
Account 1 has Rs. 0
Account 2 has Rs. -50
```

Can we ever deposit a negative amount to our Account?

Obviously NO

- All these are little **bugs** that will:
    - Allow us to extract money out using deposit()
    - Allow us to withdraw money even when balance is not enough

How can we handle these situations then?

- Our `withdraw()` method **should NOT allow the withdrawl "if `withdrawl amount < balance` "**
- Our `deposit()` method **should NOT allow depositing a negative amount**

Let's add these conditions to our custom methods

```
class Account:

    counter = 0

    def __init__(self, openingBal=0):
        Account.counter += 1
        self.id = Account.counter
        self.bal = openingBal

    def deposit(self, amount):
        if amount > 0:  # condition added to deposit
            self.bal += amount

    def withdraw(self, amount):
        if amount > 0 and self.bal >= amount:  # condition added to withdraw
            self.bal -= amount

    def __str__(self):
        return f"Account {self.id} has Rs. {self.bal}"


a1 = Account(100)
a2 = Account()
a1.deposit(50)
print(a1)
a1.withdraw(10)
print(a1)

print(a1)
print(a2)



Account 1 has Rs. 150
Account 1 has Rs. 140
Account 1 has Rs. 140
Account 2 has Rs. 0
```

Takeaway from this:

- These are some **logical things** and **bugs we need to take care of**

- We will come across these logical things in our **day-to-day programming life**

And it's not just limited to OOP

- It **applies to programming in general**

- We always need to **think critically** about the **corner cases** while writing code

- We need to take care of these logics **while modelling real-world scenarios and objects**

Conclusion:

- This is how we can:
  - **Keep creating custom methods**, and
  - **Keep enhancing functionality** of our classes

Inheritance

- Lets I want to **create different kinds of accounts**
- Savings and current account

Question: What is the difference between current and savings account?

- Savings - say, we **limit on the number of transactions** <100
- Currrent - can be any number of transactions

This is just a small difference

Does it makes sense for me to write two entire seperate classes for such a small change?

- You will be **duplicating a lot of code**
- you will have two copies
- **If you want to make any changes**, you will have to **make changes to both** - difficult to maintain

```
# same code
class Account:
    counter = 0
    def __init__(self, openingBal=0):
        Account.counter += 1
        self.id = Account.counter
        self.bal = openingBal

    def deposit(self, amount):
        if amount >= 0:
            self.bal += amount

    def withdraw(self, amount):
        if amount >= 0 and self.bal >= amount:
            self.bal -= amount

    def __str__(self):
        return f"Acc {self.id} has {self.bal}"

    def __repr__(self):
        return f"{id}"

class SavingsAccount(Account):
    pass

class CurrentAccount(Account):
    pass

sa1 = SavingsAccount()
ca1 = CurrentAccount()
```

Question: Will `numTransations` be a class variable or instance variable

- No, I would need define `self.numTrans` in the `__init__` method
- But if I **overwrite** the `__init__` method in the children class, the parent `__init__` wont be called - **overriding**
- In that case, I will to **add all the methods as well**
- We will slowly end up duplicating the code

Kills the points of having Inheritance

Lets make a trafe-off

- **Lets add `self.numTrans` in the parent code itself**

What should happen everytime we a deposit or withdraw?

`numTrans` should increase

Question: But where do we put the changes related to limit?

Maybe copy of wihdraw and deposit method in SavingAccount

But that will start **duplicating the code** - **lets minimize that**

- Lets create another variable `maxTransactions` in the parent code instead of hard-coded value 100

```
# same code
class Account:
    counter = 0
    def __init__(self, openingBal=0):
        Account.counter += 1
        self.id = Account.counter
        self.bal = openingBal
        self.numTrans = 0
        self.maxTrans = 2 # new

    def deposit(self, amount):
        # do you understand why < and not <=?
        if amount >= 0 and self.numTrans < self.maxTrans: # new
            self.bal += amount
            self.numTrans += 1 # new

    def withdraw(self, amount):
        if amount >= 0 and self.bal >= amount and self.numTrans < self.maxTrans
            self.bal -= amount
            self.numTrans += 1 # new

    def __str__(self):
        return f"Acc {self.id} has {self.bal}"

    def __repr__(self):
        return f"{id}"

class SavingsAccount(Account):
    pass

class CurrentAccount(Account):
    pass

sa1 = SavingsAccount()
ca1 = CurrentAccount()
print(sa1)
sa1.deposit(100)
print(sa1)
sa1.deposit(100)
print(sa1)
sa1.deposit(100)
print(sa1) # we can see that deposit doeesn't happen further
```

```
Acc 1 has 0
Acc 1 has 100
Acc 1 has 200
Acc 1 has 200
```

So, we see that

- Transactions beyond 2 got ignored

- Even **if we allow max 3 transaction by making** `self.maxTrans = 3`
    - All transactions beyond 3 will get ignored

But what is the problem here?

- We are **making changes in parent class** code
- Due to this, **both Savings Account and Current Account are getting affected**

Let's check this

```
ca1.deposit(100)
print(ca1)
ca1.deposit(100)
print(ca1)
ca1.deposit(100)
print(ca1)
```

```
Acc 2 has 100
Acc 2 has 200
Acc 2 has 200
```

How do we resolve this issue?

How do we ensure that different `maxTrans` limits get applied to `SavingsAccount` and `CurrentAccount` ?

- Let's say `maxTrans = 2` for `SavingsAccount` and `5` for `CurrentAccount`

- Just **update the value of** `maxTrans` from `2` to `5` **for Current Account**

- This will fix the issue

How can we do that?

- Inside child class `CurrentAccount`

```
# same code
class Account:
    counter = 0
    def __init__(self, openingBal=0):
        Account.counter += 1
        self.id = Account.counter
        self.bal = openingBal
        self.numTrans = 0
        self.maxTrans = 2

    def deposit(self, amount):
        if amount >= 0 and self.numTrans < self.maxTrans:
            self.bal += amount
            self.numTrans += 1

    def withdraw(self, amount):
        if amount >= 0 and self.bal >= amount and self.numTrans < self.maxTrans
            self.bal -= amount
            self.numTrans += 1

    def __str__(self):
        return f"Acc {self.id} has {self.bal}"

    def __repr__(self):
        return f"{id}"

class SavingsAccount(Account):
    pass

class CurrentAccount(Account):
    self.maxTrans = 5  # new

sa1 = SavingsAccount()
ca1 = CurrentAccount()
print(sa1)
sa1.deposit(100)
print(sa1)
sa1.deposit(100)
print(sa1)
sa1.deposit(100)
print(sa1)
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-50-e2f5e59fa50b> in <module>
     28     pass
     29
---> 30 class CurrentAccount(Account):
     31     self.maxTrans = 5  # new
     32


<ipython-input-50-e2f5e59fa50b> in CurrentAccount()
     29
     30 class CurrentAccount(Account):
---> 31     self.maxTrans = 5  # new
     32
     33 sa1 = SavingsAccount()


NameError: name 'self' is not defined
```

Now why did we receive this Error?

- `self` was defined for parent `Account` class
- There is no self for child `CurrentAccount` class
- So, we'd have to define a `__init__` inside `CurrentAccount` class

```
# same code
class Account:
    counter = 0
    def __init__(self, openingBal=0):
        Account.counter += 1
        self.id = Account.counter
        self.bal = openingBal
        self.numTrans = 0
        self.maxTrans = 2

    def deposit(self, amount):
        if amount >= 0 and self.numTrans < self.maxTrans:
            self.bal += amount
            self.numTrans += 1

    def withdraw(self, amount):
        if amount >= 0 and self.bal >= amount and self.numTrans < self.maxTrans
            self.bal -= amount
            self.numTrans += 1

    def __str__(self):
        return f"Acc {self.id} has {self.bal}"

    def __repr__(self):
        return f"{id}"

class SavingsAccount(Account):
    pass

class CurrentAccount(Account):
    def __init__(self):  # new
        self.maxTrans = 5



sa1 = SavingsAccount()
ca1 = CurrentAccount()

print(sa1)
sa1.deposit(100)
print(sa1)

print(ca1)  # new
ca1.deposit(100)  # new
print(ca1)  # new
```

```
Acc 1 has 0
Acc 1 has 100



---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call last)

<ipython-input-51-17bc15c9d11a> in <module>
     41 print(sa1)
     42
---> 43 print(ca1)  # new
     44 ca1.deposit(100)  # new
     45 print(ca1)  # new


<ipython-input-51-17bc15c9d11a> in __str__(self)
     20
     21     def __str__(self):
---> 22         return f"Acc {self.id} has {self.bal}"
     23
     24     def __repr__(self):


AttributeError: 'CurrentAccount' object has no attribute 'id'
```

Using `super()` method

- We use an **in-built helper method** **`super()`**
- It **represents the parent class**
- `super()` **gives access** to **instance** of **parent class**

`super().__init__()` calls `__init__` method of parent class

- Don't have to pass `self` here
- Because `self` has already gone through `super()`

```
# same code
class Account:
    counter = 0
    def __init__(self, openingBal=0):
        Account.counter += 1
        self.id = Account.counter
        self.bal = openingBal
        self.numTrans = 0
        self.maxTrans = 2

    def deposit(self, amount):
        if amount >= 0 and self.numTrans < self.maxTrans:
            self.bal += amount
            self.numTrans += 1

    def withdraw(self, amount):
        if amount >= 0 and self.bal >= amount and self.numTrans < self.maxTrans
            self.bal -= amount
            self.numTrans += 1

    def __str__(self):
        return f"Acc {self.id} has {self.bal}"

    def __repr__(self):
        return f"{id}"

class SavingsAccount(Account):
    pass

class CurrentAccount(Account):
    def __init__(self):
        super().__init__()  # new
        self.maxTrans = 5


sa1 = SavingsAccount()
ca1 = CurrentAccount()

print(sa1)
sa1.deposit(100)
print(sa1)

print(ca1)
ca1.deposit(100)
print(ca1)
```

```
Acc 1 has 0
Acc 1 has 100
Acc 2 has 0
Acc 2 has 100
```

- Now we can see the error has gone away

Now Let's test our code for different `maxTrans` for `SavingsAccount` and `CurrentAccount`

- We'll keep `maxTrans = 2` in parent `Account` class
    - It will be inhereted by `SavingsAccount`
- We'll set `maxTrans = 3` for `CurrentAccount`

```
# same code
class Account:
    counter = 0
    def __init__(self, openingBal=0):
        Account.counter += 1
        self.id = Account.counter
        self.bal = openingBal
        self.numTrans = 0
        self.maxTrans = 2

    def deposit(self, amount):
        if amount >= 0 and self.numTrans < self.maxTrans:
            self.bal += amount
            self.numTrans += 1

    def withdraw(self, amount):
        if amount >= 0 and self.bal >= amount and self.numTrans < self.maxTrans
            self.bal -= amount
            self.numTrans += 1

    def __str__(self):
        return f"Acc {self.id} has {self.bal}"

    def __repr__(self):
        return f"{id}"

class SavingsAccount(Account):
    pass

class CurrentAccount(Account):
    def __init__(self):
        super().__init__()
        self.maxTrans = 3  # new


sa1 = SavingsAccount()  # max 2 transactions allowed
ca1 = CurrentAccount()  # max 3 transactions allowed

print(sa1)
sa1.deposit(100)
sa1.withdraw(50)
sa1.deposit(100)  # will not happen
sa1.withdraw(50)  # will not happen
print(sa1)

print(ca1)
ca1.deposit(100)
ca1.deposit(100)
ca1.deposit(100)
ca1.deposit(100) # will not happen
print(ca1)
```

```
Acc 1 has 0
Acc 1 has 50
Acc 2 has 0
Acc 2 has 300
```

Private Attributes

What if we manually set balance of an account?

```
# same code
class Account:
    counter = 0
    def __init__(self, openingBal=0):
        Account.counter += 1
        self.id = Account.counter
        self.bal = openingBal
        self.numTrans = 0
        self.maxTrans = 2

    def deposit(self, amount):
        if amount >= 0 and self.numTrans < self.maxTrans:
            self.bal += amount
            self.numTrans += 1

    def withdraw(self, amount):
        if amount >= 0 and self.bal >= amount and self.numTrans < self.maxTrans
            self.bal -= amount
            self.numTrans += 1

    def __str__(self):
        return f"Acc {self.id} has {self.bal}"

    def __repr__(self):
        return f"{id}"

class SavingsAccount(Account):
    pass

class CurrentAccount(Account):
    def __init__(self):
        super().__init__()
        self.maxTrans = 3


sa1 = SavingsAccount()
ca1 = CurrentAccount()

print(sa1)
sa1.deposit(100)
sa1.withdraw(50)
sa1.deposit(100)
sa1.withdraw(50)
sa1.bal = 999999999  # new - manually setting the balance
print(sa1)

print(ca1)
ca1.deposit(100)
ca1.deposit(100)
ca1.deposit(100)
ca1.deposit(100)
print(ca1)
```

```
Acc 1 has 0
Acc 1 has 999999999
Acc 2 has 0
Acc 2 has 300
```

How can we make some variable private in a Python class?

- Simply use `__` **(double underscore)** before variable's name
- **Example**: `self.__bal`
- Single underscore (`self._bal`) **indicates other developers** that you **should not change it outside the class**
    - Can change it, but should not
- Double underscore (`self.__bal`) **enforces** that it **cannot be changed outside the class**

```
# same code
class Account:
    counter = 0
    def __init__(self, openingBal=0):
        Account.counter += 1
        self.id = Account.counter
        self.__bal = openingBal  # new --> self.__bal
        self.numTrans = 0
        self.maxTrans = 2

    def deposit(self, amount):
        if amount >= 0 and self.numTrans < self.maxTrans:
            self.__bal += amount  # new --> self.__bal
            self.numTrans += 1

    def withdraw(self, amount):
        if amount >= 0 and self.__bal >= amount and self.numTrans < self.maxTra
            self.__bal -= amount  # new --> self.__bal
            self.numTrans += 1

    def __str__(self):
        return f"Acc {self.id} has {self.__bal}"  # new --> self.__bal

    def __repr__(self):
        return f"{id}"

class SavingsAccount(Account):
    pass

class CurrentAccount(Account):
    def __init__(self):
        super().__init__()
        self.maxTrans = 3


sa1 = SavingsAccount()
ca1 = CurrentAccount()

print(sa1)
sa1.deposit(100)
sa1.withdraw(50)
sa1.deposit(100)
sa1.withdraw(50)
sa1.__bal = 999999999  # new - Manual change will NOT work
print(sa1)

print(ca1)
ca1.deposit(100)
ca1.deposit(100)
ca1.deposit(100)
ca1.deposit(100)
print(ca1)
```

```
Acc 1 has 0
Acc 1 has 50
Acc 2 has 0
Acc 2 has 300
```

- As we see, `sa1.bal` did NOT change to `999999999`

- Now **directly changing value of `sa1.bal` did NOT work**

Privacy is enforced

- Now if we want to change `bal`, we **have to go through** `deposit()` or `withdraw()`

- **Solves the problem of someone else trying to access `bal` directly** outside the class

Polymorphism

- One last topic in OOP before we wind up today's lecture

- **Polymorphism means having many forms**

- It's the **ability of an object to be displayed in more than one form**

Let's take our previous example and modify it

- Parent class `Account` with child classes `SavingsAccount` and `CurrentAccount`

- Suppose our application needs **methods** to **calculate interest** for each specific account

- The **interest of each account is calculated differently**

- So we **can't have a single implementation.**

What can we do?

- Well we **could throw in separate methods in each class**
    - `getSavingsInterest()`, `getCurrentInterest()` etc...

- But this makes it **harder to remember each method's name.**

We can make things simpler with polymorphism

How will we implement this?

- **Parent class declares a function without providing an implementation**.

- **Each child class inherits the function declaration** and can **provide its own implementation**

- Let's give `Account` class a method called `getInterest()` - which is **inherited by both child classes**.

- With polymorphism, each **child class** may have its **own way of implementing the method**.

```
class Account:

    counter = 0

    def __init__(self, openingBal=0):
        Account.counter += 1
        self.id = Account.counter
        self.bal = openingBal
        self.numTrans = 0
        self.maxTrans = 2

    def deposit(self, amount):
        if amount >= 0 and self.numTrans < self.maxTrans:
            self.bal += amount
            self.numTrans += 1

    def withdraw(self, amount):
        if amount >= 0 and self.bal >= amount and self.numTrans < self.maxTrans
            self.bal -= amount
            self.numTrans += 1

    def getInterest(self): # new
        pass


    def __str__(self):
        return f"Acc {self.id} has {self.bal}"  # new --> self.__bal

    def __repr__(self):
        return f"{id}"

class SavingsAccount(Account):
    def __init__(self):
        super().__init__()

    def getInterest(self):  # new - Interest calculation for Savings Account
        interest = self.bal*0.07
        print(f"Interest on Account {self.id} is {interest}")


class CurrentAccount(Account):
    def __init__(self):
        super().__init__()
        self.maxTrans = 3

    def getInterest(self):  # new - Interest calculation for Current Account
        interest = (self.bal*0.05)/self.numTrans
        print(f"Interest on Account {self.id} is {interest}")


sa1 = SavingsAccount()
ca1 = CurrentAccount()
```

```
print(sa1)
sa1.deposit(100)
sa1.withdraw(50)
print(sa1)
sa1.getInterest()


print(ca1)
ca1.deposit(100)
ca1.deposit(100)
ca1.deposit(100)
print(ca1)
ca1.getInterest()
```

```
Acc 1 has 0
Acc 1 has 50
Interest on Account 1 is 3.5000000000000004
Acc 2 has 0
Acc 2 has 300
Interest on Account 2 is 5.0
```

So our `getInterest()` method has “many forms” - This is Polymorphism!

- Having specialized implementations of the same methods for each class.

So, What does polymorphism achieve?

- In effect, polymorphism cuts down the work of the developer.

- When time comes to create more specific child classes with certain unique attributes and behaviors,
    - developer can alter the code in the specific areas where the responses differ.
- All other pieces of the code can be left untouched.