CIS STUDY SHEET COMPILATION

- Each Topic Has:
  - Notes on that topic
  - Examples
  - Process of Attack
  - Relating back to projects (maybe?)

- List of Topics:
  - Classes and objects
  - Subclass
  - Listeners
  - Base Case / OOP Recursion /Algorithms
    - Loop design
    - Recursion (in object orientation and subclassing)
    - Plan composition including multi-pass algorithms
  - Memory Model / Aliasing
  - Regular Expressions (Regex)
  - Masking / Bit Manipulation

- Challenges:
  - RandomTree Challenge (lab)
  - Node Challenge (lab)

Classes and Objects:

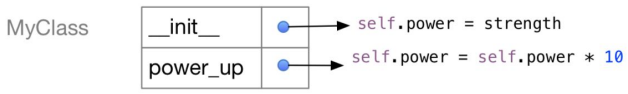
- Class vs. object
  - Fields are in the object
  - Methods are in the class
  - Object has reference to class

- Inherited & overridden methods
  - Object has reference to class
  - Class has reference to methods
  - Inherited methods are copied references
  - Overridden methods are references that aren't shared with superclass

```
class MyClass(object):

    def __init__(self, strength: int):
        self.power = strength

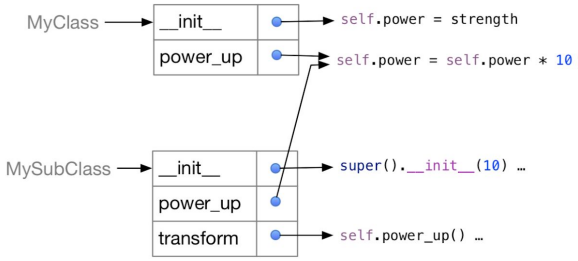
    def power_up(self):
        self.power = self.power * 10
```




```
class MySubClass(MyClass):

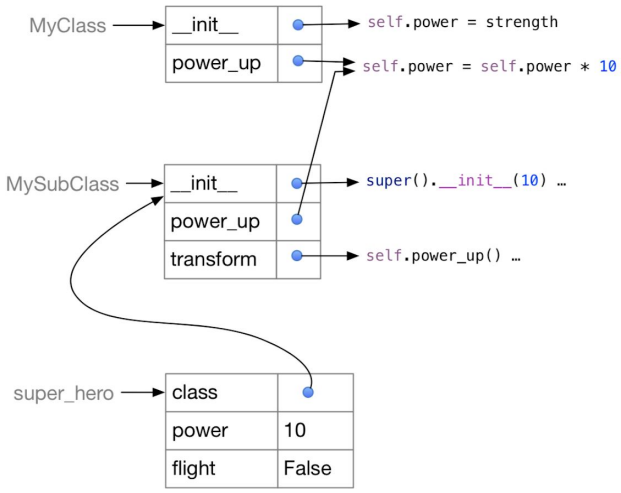
    def __init__(self):
        super().__init__(10)
        self.flight = False

    def transform(self):
        self.power_up()
        self.flight = True
```



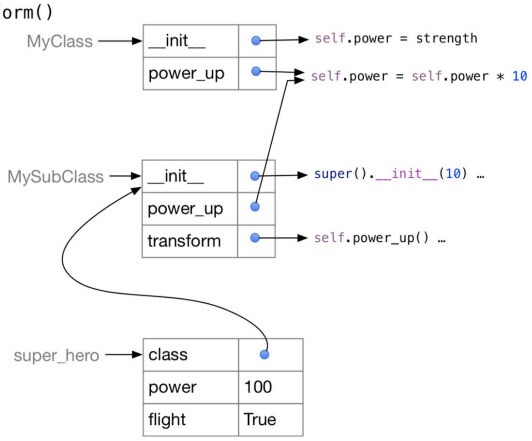
Object

```
super_hero = MySubClass()
```



Method call (dynamic dispatch)

```
super_hero.transform()
```



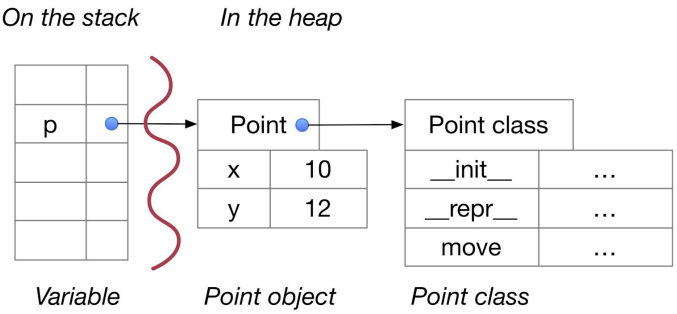
Alyssa's Notes (Classes and Objects)

- Why classes / objects / methods?
  - Used in most modern programming languages.
  - It allows us to logically group our data functions in a way that is easy to use / reuse and also easy to build upon if need be.
  - Object is a thing that has data and contains methods. It has built in attributes to describe it.
    - alyssa.female == attribute to the object
    - alyssa.study() == method to the object
  - Data = attributes , Functions = methods
    - methods are just like functions but they are inside the class and uses the data from that same class for that object instance.
  - self is a placeholder for the objects

Example:

```
class Employee:
pass
```

If left empty = error, would need "pass" to skip over it.



Our class is basically a blueprint for creating instances and each unique (employee in this case) will be an instance of that class. Ex) class Employee would have self and the different employees (Bob, Jim, Pam) would be the instances of the Employee class.

```
emp_1 = Employee()
emp_2 = Employee()
```

* These are both employee objects and they are both unique. *

- Instance variables: contain data that is unique to each instance.

```
emp_1.first = "Alyssa"
emp_1.last = "Kelley"
emp_1.email = "Alyssa.Kelley@company.com"
emp_1.pay = 50000
```

This is manually creating this information for each employee being those instance variables. This uses a lot of code and prone to mistakes so it is not a good use of classes.

```
def __init__(self):
```

initialize / constructor method

When we create methods within a class they receive the instance as the first argument. Technically you can call this instance whatever you want, but good style is referring to it as "self" and you can add whatever other information you want to collect as parameters. Note: that when you are in the class, self has a **type** and the type is that class. So in this case, self has the type of "Employee" because we are in class Employee.

```
def __init__(self, first, last, pay):
```

```
self.first = first
self.last = last
self.pay = pay
self.email = first + ".” last + "@company.com”
```

The underlined section does not technically need to be the same.
It could `self.fname = first` but good style is to keep it them same.

This is the same thing as saying “emp_1.first = “Alyssa” (etc..) but now it is being done automatically instead of manually when we create our Employee objects.

So now when we create our instances of our Employee class we can pass through the values that we specified in our ___init___ method instead of just creating the object as `emp_1 = Employee()`.

When we create the employee done in `emp_1 = Employee('Alyssa', 'Kelley', 50000)`, it passes along this instance (self) automatically so there is no need to specify this as a parameter. The order in which the information is passed along when creating an object is IMPORTANT!

```
emp_1 = Employee('Alyssa', 'Kelley', 50000)
emp_2 = Employee('Test', 'User', 60000)
```

self — class — self.first = first — self.last = last — self.pay = pay

Name, email and pay are all attributes of the Employee class. If you want the ability to perform an action (such as display the full name) then you would need to add some methods to the class.

```
print('{} {}'.format(emp_1.first, emp_1.last)
```

This is a manual way to display the employee's full name, but we can create a method that does this automatically (better).

```
class Employee:
    def __init__:
        **SEE ABOVE**
    def fullname(self):
        return ('{} {}'.format(self.first, self.last)
print(emp_1.fullname())
```

Must include (self) b/c it passes automatically.

Need the parenthesis here because this is a method and not an attribute. If you leave the () off then it would print the method instead of return the value of the method.

```
Employee.fullname(emp_1)
emp_1.fullname()
```

* These two do the exact same thing!
*

- Class variables: variables that are shared among all instances of a class.
  - Instance variables can be unique for each instance like our names, email, and pay.
  - Class variables should be the same for each instance. Ex) annual raise increase for all employees.

```
class Employee:
     def __init__(self):
          ** SEE ABOVE **
     def fullname(self):
          ** SEE ABOVE **
     def apply_raise(self):
          self.pay = int(self.pay * 1.04)  # 4% annual raise
print(emp_1.pay)  # would return 50000
emp_1.apply_raise()  # this is actually make the method occur
print(emp_1.pay)  #
```

Note: this is a bad way of coding because it is considered "hard coding" and it would be better to have the percent that you want to be used be a seperate variable so it can easily be changed in the future with no logic errors. (see below for a better way)

```
class Employee:
     raised_amount = 1.04
     def __init__(self): ...
     def fullname(self): ...
     def apply_raise(self):
          self.pay = int(self.pay * self.raised_amount)
```

If you left off "self" from "self.raised_amount" we would get a NameError. You need to access this variable either through the class itself (Employee.raised_amount) or through an instance of the class as seen here (self.raised_amount). Even though this is a class variable, it is accessed through our instance.

```
print(emp_1.raised_amount)
```
Access via class.

```
print(Employee.raise_amount)
```
Accessed via instance and since it doesn't have this attribute itself, it accesses the classes attribute.

```
print(emp_2.raise.amount)
```

Note: you can check the namespaces to see if it is a class variable or not. If you checked the namespace for the instance of the class (emp_1) then it would contain the variables that are conducted in the ___init___ method and passed in. If you check the class namespace, then it would have things like the raised_amount and it's value. How to do this:
`print(instance.__dict__)` or `print(class.__dict__.`

- You can change the value of a class variable by doing the following:

```
Employee.raised_amount = 1.05
```

- You can also just change it for that instance instead of the whole class:

```
emp_1.raised_amount = 1.05
```
This would then add this variable and amount to the namespace for emp_1, but none of the other instances (like emp_2) would have this variable or value in their namespace. Note: The only way this is possible to do is because we added raised_amount as a class variable and had it in the method as self.raised_amount and not Employee.raised_amount. Using self here allows any subclass to override that constant if they wanted to.

Another example of when a class variable would be best (and the only logical options) is with the number of employees.

```
class Employee:
     number_of_emps = 0  # initializing it to be incremented.
     def __init__(self, first, last, pay):
          self.first = first
          self.last = last
          self.pay = pay
          self.email = first + '.' + last + '@compnay.com'
          Employee.number_of_emps += 1
```

Would definitely want to use the class variable of this so it cannot be overridden.

```
print(Employee.number_of_emps)  # would return 2 (emp_1, emp_2)
```

To sum it up... class variables (Employee.number_of_emps) and this cannot be overridden and it is the same for all instances; instance variables (self.amount_raised) can be overridden and it is possible for their value to be different for each and any instance.

- Regular methods in a class automatically take the instance as the first argument (self)
- Class methods take the class as the first argument
    - ```
      def set_raise_amount(cls,amount)
            cls.raise_amt = amount
      ```
- Static methods don't pass anything automatically: no instance pass and no class passed. This means they behave just like a regular function except we include them in our classes b/c they have some logical connection with the class.

Note: we did not really go over this on class so I am skipping the different between @classmethods and @staticmethods.

- Special (Magic/Dunder) Methods:
    - These special methods allow us to emulate some built_in behavior within Python and it's also how we implement operator overloading.
    - These special methods are always surrounded by double underscores (dunder).
    - Examples: ___init___, ___repr___, ___str___
- ___init___
    - Sets all of our attributes. Called the constructor for a class.
- ___repr___
    - An unambiguous representation of the object and should be used for debugging and logging meant to be seen by other developers.
    - You should always have a ___repr___ method because if you have this and no ___str___ then calling str(...) on an object (employee) then it will just use the ___repr___ as a fallback so this is a good minimum to have.
      ```

- ___str___
  - A more readable representation of an object and is meant to be used as a display to the end-user.

```
class Employee:
     def __init__(self, first, last, pay):
     def fullname(self, first, last):   ** SEE ABOVE FOR THIS **
     def apply_raise(self):

     def __repr__(self):
          return "Employee('{}', '{}', '{}')".format(self.first,
               self.last, self.pay)
emp_1 = Employee('Alyssa', 'Kelley', 50000)
print(emp_1)
```

This now prints Employee('Alyssa', 'Kelley', 50000) instead of the location of this employee. The __repr__ is used as something that can be used in Python meaning that you can now copy this printed representation of emp_1 and copy it and use it with no error.

```
          def __str__(self):
               return "{} - {}".format(self.fullname(),
                    self.email)
print(emp_1)
```

Alyssa Kelley - Alyssa.Kelley@company.com is returned and this is now what will be printed by default for emp_1.

`print(repr(emp_1))` This is how we can still access the repr representation.

`print(str(emp_1))` How to specifically access the str representation.

Note: calling `repr(emp_1) == emp_1.__repr__()` and same thing for all of the special methods.

- There are other special methods that are used more specifically for arithmetic.
  - Examples:
    - `print(1+2) == print(int.__add__(1, 2))`
    - `print('a', 'b') == print(str.__add__('a', 'b')`
  - And if you want, you can customize these to work for our objects by creating a dunder add (___add___) method.

In our employee example, I am going to create an __add__ method to add the employees together and get their gross salary

```
class Employee:
 ...
          right side of operation      left side of operation
     def __add__(self,other):
          return self.pay + other.pay   assuming both are Employee objects
print(emp_1 + emp_2)  # would return 110000 (50000 + 60000)
```

If we tried to do this and we didn't have our custom __add__ method then we would get a TypeError saying it didn't know how to add 'Employee' and 'Employee'

Now we are going to create __len__ to return the total characters in their full name

```
    def __len__(self):
        return len(self.fullname())
print(len(emp_1))  # This would return 13
```

Deeper dive into classes and objects: Class is basically a table of functions. Subclass inherits by copying the table. Subclass overrides by replacing table entries. Object has a reference to the class. Method call o.foo() is really o.___class___.foo(o) (object o becomes the "self" argument)

Note: He will give us a problem where the superclass is calling methods in the subclass (look at eval from calculator project and the compiler project with binop having gen but not knowing what thing to do (ADD etc)

LAB EXAMPLE:
The point of this program is that it is about music and there is a class for a Collection, an Artist, their Album, and their Songs on that album. The goal is to find the artist in the collection that has the shortest average song duration for their entire album. Point of attack: you will need to loop through the entire artists and then all of their albums, and all of their songs on each album. You will need to keep track of the list of their songs, and the average song duration for the songs on the album and the artist that is the shortest.

```
class Collection:
...etc..

def shortest_avg_artist(self):
    short_avg = math.inf # infinity is always bigger than anything
    short_artist = None
    for artist in self.artists:
        song_list = []
        for album in artist.albums:
            for song in album.songs:
                song_list.append(duration)
            avg = sum(song_list) / len(song_list)
            if avg < short_avg:
                short_avg = avg
                short_artist = artist
    return short_artist
```

Practice Exam Questions focused on understanding CLASSES AND OBJECTS:

Midterm (Class and Objects):

Question Type: What does this print?

```
class Kiosk(object):
    """A bike share station with a limited number of racks."""

    def __init__(self, capacity: int):
        self._capacity = capacity
        self._occupied = 0

    def bikes_available(self) -> int:
        """How many bikes available to rent?"""
        return self._occupied

    def space_available(self) -> int:
        """How many empty racks for returning bikes?"""
        return self._capacity - self._occupied

    def park_bike(self) -> bool:
        """Park bike and return True iff there is an empty rack."""
        if self.space_available() > 0:
            self._occupied += 1
            return True
        return False

    def take_bike(self) -> bool:
        """Take bike and return True iff there is an occupied rack."""
        if self.bikes_available() > 0:
            self._occupied -= 1
            return True
        return False

kiosk = Kiosk(2)

kiosk.park_bike()
kiosk.park_bike()
kiosk.park_bike()

kiosk.take_bike()
kiosk.take_bike()

print("{} bicycles available to rent".format(kiosk.bikes_available()))
```

What does it print?

0 bicycles available to rent.
(Notice that the third attempt to park a bike has no effect.)

Practice Midterm (Classes and Objects):

Question Type: What does this print?

```
class Circle(object):

    def __init__(self, x, y, radius):
        self.x = x
        self.y = y
        self.radius = radius

    def move(self, dx, dy):
        self.x = self.x + dx
        self.y = self.y + dy

    def expand(self, factor):
        self.radius = self.radius * factor

    def __repr__(self):
        return "Circle({}, {}, {})".format(self.x, self.y, self.radius)

    def __str__(self):
        return repr(self)

def q1():
    c = Circle(5, 3, 15)
    c.expand(4)

    d = Circle(-1, -3, 4)
    d.move(15,15)

    print("c: {}".format(c))
    print("d: {}".format(d))
```

What does q1( ) print?

c: Circle(5, 3, 60)
d: Circle(14, 12, 4)

Question Type: Finish this method.

6. [10 points] Finish the pour_out method of class Bucket below so that the test cases pass.

```
class Spillage(Exception):
    pass

class Bucket(object):
    """You can put stuff in!  You can pour stuff out!"""

    def __init__(self, capacity: int) -> None:
        self.capacity = capacity
        # Initially the bucket is empty
        self.holding = 0

    def pour_in(self, amount: int) -> None:
        """Pour amount of liquid into the bucket,
         if there is enough room left.
         """
        space = self.capacity - self.holding
        if amount > space:
            raise Spillage("Too much!  Overflowing!")
        self.holding += amount

    def pour_out(self, requested_amount: int) -> int:
        """Pour UP TO amount of liquid from the
        bucket.  If more is requested than the bucket
        currently holds, pour out just what the bucket
        currently holds.  Returns the amount of liquid
        actually poured out.  See tests below for
        examples.
        """
        #Sample Answer
        pour_amount = min(self.holding, requested_amount)
        self.holding -= pour_amount
        return pour_amount


b = Bucket(20)
b.pour_in(10)
out = b.pour_out(7)
assert out == 7
out = b.pour_out(7)
assert out == 3
out = b.pour_out(7)
assert out == 0
```

Practice Final (Classes and Objects):

Question Type: What does this print?

```
class Interval(object):
    """An interval has a starting point and an ending point"""

    def __init__(self, start: int, end: int):
        self.start = start
        self.end = end

    def __repr__(self):
        return "Interval({},{})".format(self.start, self.end)

    def __str__(self):
        return self.__repr__()

    def slide(self, delta: int):
        self.start += delta
        self.end += delta

    def scale(self, factor: int):
        self.end = self.start + factor * (self.end - self.start)


from_zero = Interval(0,1)
from_one = Interval(1,2)
from_zero.slide(10)
from_zero.scale(10)
from_one.slide(8)
print(from_zero)
print(from_one)
```

What does it print?

Interval(10,20)
Interval(9,10)

Most people did well on this problem. If you got Interval(1,9) or Interval(20,21), I guessed you probably applied the wrong operation (slide vs scale) and took off 3 points. If you didn't use the `__repr__` function took off 3. I took off a single point for an off-by-one error or missing parentheses.

Practice Final (Classes and Objects):

Question Type: Finish this method.

4. [10 points] Complete method `maximal` of class `SegList` so that this program produces the expected output, `Seg(1,10)`.

```
from typing import List

class Seg(object):
    def __init__(self, low: int, high: int):
        self.low = low   # low attribute is public, read-only
        self.high = high # high attribute is public, read-only

    def length(self):
        return self.high - self.low

    def __str__(self):
        return "Seg({},{})".format(self.low, self.high)

class Comparison(object):
    def compare(self, s1: Seg, s2: Seg) -> bool:
        raise NotImplementedError("compare method must be overridden")

class Longer(Comparison):
    def compare(self, s1: Seg, s2: Seg) -> bool:
        return s1.length() > s2.length()

class SegList(object):
    def __init__(self, segs: List[Seg]):
        assert len(segs) > 0, "segs must not be empty"
        self._contents = segs

    def maximal(self, comparison: Comparison) -> Seg:
        """Return seg that is maximal according to comparison"""
         max = self._contents[0]
        for seg in self._contents:
            if comparison.compare(seg, max):
                max = seg
        return max


segs = SegList([Seg(0,3), Seg(1,2), Seg(1,10), Seg(5,6)])
print(segs.maximal(Longer())) # Expected output: Seg(1,10)
```

The panoply of issues I saw on this one defy summarization. The most common issues were not using `comparison.compare` to determine the maximum (e.g., always using length), for which I deducted 5 points, or returning a string when the method header declares that a Seg object should be returned. Picking an extreme element (min, max, whatever) is such a common idiom that I did not expect you to have trouble with it, but I saw some crazy algorithms that will not work.

Classes and Object -> Subclassing/Polymorphism:
Alyssa's Notes (Subclassing)

- Inheritance allows us to inherit attributes and methods from a parent class and this is useful b/c we can create subclasses and get all of the functionality of our parent class and then we can overwrite or add completely new functionality without affecting the parent class in any way.

```
class Developer(Employee):
    pass
```
Inherits from the Employee class

Without even adding any code at all to this subclass, we still get all of the information from the class Employee that we already created.

```
dev_1 = Developer('Alyssa', 'Kelley', 50000)
dev_2 = Developer('Test', 'User', 60000)
print(dev_1.email)   # returns Alyssa.Kelley@company.com
print(dev_2.email)   # returns Test.User@company.com
```

When we instantiated our developers, it first looked in our developer class for our ___init___ method and it's not going to find it within our developer class b/c it is currently empty so Python then is going to go up this chain of inheritance until it finds what it's looking for and this chain is called the method resolution order.

```
print(help(Developer))
```
This will show the method resolution where Python searched for attributes and methods.

Order: Developer
Employee
builtins.object
... it will then show you the methods and attributes all listed out and where they come from.

```
print(dev_1.pay)   # 50000 is returned
dev_1.apply_raise()   # this applies the raise
print(dev_2.pay)   # 52000
```
Now changing it so the developers get 10% raise instead of the set 4% raise...

```
class Employee:
    raise_amount = 1.10
```
By changing the `raised_amount` in this Developer subclass, it didn't change or have any effect on any of our employee instances so they all still have that `raised_amount = 1.04`

Now going to add code to pass along the developers favorite programming language. Our Employee class only passes along first, last, and pay so this means we are going to have to give the Developer class it's own __init__ method.

```
class Developer:
    raise_amount = 1.10
    def __init__(self, first, last, pay, prog_lang):
```

```
            super().__init__()
super().__init(first, last, pay)
            self.prog_lang = prog_lang
dev_1 = Developer('Alyssa', 'Kelley', 50000, 'Python')
dev_2 = Developer('Test', 'User', 60000, 'Java')

print(dev_1.email)  # Alyssa.Kelley@company.com is returned
print(dev_1.prog_lang)  # Python is returned
```

This lets out superclass Employee handle the other arguments, to specify particular ones to inherit.

Now creating a Manager subclass and going to give the option of listing out all of the employees a specific manager supervises

```
class Manager(Employee):
      def __init__(self, first, last, pay, employees = None):
            super.__init__(first, last, pay)
            if employees is None:
                  self.employees = []
            else:
                  self.employees = employees
```

Note: You NEVER want to pass in mutable data types (like a List/[]) as a default argument so this sets the employees list as empty if the argument is not provided, but if it is, then it would be that list of employees.

Adding the option to add/remove employees that the managers supervises:

```
      def add_emp(self, emp):
            if emp not in self.employees:
                  self.employees.append(emp)
      def remove_emp(self, emp):
            if emp in self.employees:
                  self.employees.remove(emp)
```

Adding a method to print out employee list...

```
def print_emp(self):
      for emp in self.employees:
            print('-->', emp.fullname())
```

This is the method created in Employee class

- How to check inheritance:

- "isinstance" is a builtin and it will tell us if an object is an instance of a class or not.

```
print(isinstance(dev_1, Developer)  # returns True
print(isinstance(dev_1, Employee)  # returns True
print(isinstance(dev_1, Manager)  # returns False
```

So even though both Developer and Manager inherit from class Employee, they aren't part of each other's inheritance.

- "issubclass" is also a builtin function that will tell us if a class is a subclass of another or not.

```
print(issubclass(Developer, Employee))  # returns True
print(issubclass(Manager, Employee))  # returns True
print(issubclass(Manager, Developer))  # returns False
```

Examples from Piazza for SUBCLASSING:

```
# Animals.py

class Animal(object):

    def __init__(self, animal_size: str, animal_age: int):
        self.multiCellular = True
        self.cellType = "Eukaryotic"
        self.kingdom = "Animalia"
        self.size = animal_size
        self.age = animal_age

class Jaguar(Animal):

    def __init__(self, jag_size: str, jag_age: int, jag_spots: bool):
        super().__init__(jag_size, jag_age)
        self.genus = "Panthera"
        self.spots = jag_spots

class Black_Panther(Jaguar):

    def __init__(self, bp_size: str, bp_age: int):
        super().__init__(bp_size, bp_age, False)
        self.furColor = "Black"

class Spotted_Jaguar(Jaguar):

    def __init__(self, sj_size: str, sj_age: int):
        super().__init__(sj_size, sj_age, True)
        self.furColor = "Orange"

print("")

elephantSam = Animal("Very Large", 20)
print("Elephant Sam: " + elephantSam.kingdom)
# Inheritance is NOT reciprocal (does not go both ways)!
# print("Elephant Sam: " + elephantSam.genus)

print("")

jaguarBob = Jaguar("Medium", 30, False)
print("Jaguar Bob: " + str(jaguarBob.age))
print("Jaguar Bob: " + jaguarBob.cellType)
print("Jaguar Bob: " + jaguarBob.genus)

print("")

jaguarAlice = Black_Panther("Large", 25)
print("Jaguar Alice: " + jaguarAlice.size)
print("Jaguar Alice: " + jaguarAlice.genus)
print("Jaguar Alice: " + str(jaguarAlice.spots))
print("Jaguar Alice: " + jaguarAlice.furColor)

print("")

jaguarMary = Spotted_Jaguar("Small", 25)
print("Jaguar Mary: " + jaguarMary.size)
print("Jaguar Mary: " + jaguarMary.genus)
print("Jaguar Mary: " + str(jaguarMary.spots))
print("Jaguar Mary: " + jaguarMary.furColor)
```

Examples from Piazza for SUBCLASSING:

```
class Animal(object):

    def __init__(self):
        self.multiCellular = True
        self.cellType = "Eukaryotic"
        self.kingdom = "Animalia"

class Mother(Animal):

    def __init__(self):
        super().__init__()
        self.eyeColor = "Brown"

class Father(Animal):

    def __init__(self):
        super().__init__()
        self.hairColor = "Purple"

class Child(Mother, Father):

    def __init__(self):
        super().__init__()


print("")

alice = Mother()
print("Alice Eyes: " + alice.eyeColor)

print("")

bob = Father()
print("Bob Hair: " + bob.hairColor)

print("")

mary = Child()
print("Mary Eyes: " + mary.eyeColor)
print("Mary Hair: " + mary.hairColor)
print("Mary MultiCellular?: " + str(mary.multiCellular))
print("Mary Cell Type: " + mary.cellType)
print("Mary Kingdom: " + mary.kingdom)
```

Practice Exam Questions focused on understanding SUBCLASSING:

Midterm (Subclassing):

Question Type: Finish this method.

4. [10 points] Implement classes Sheep and Dog as subclasses of Animal, without overriding method `speak`. You do not need to implement `__str__`, `__repr__`, etc; write the minimum code required to get the expected output.

```
class Animal(object):
    """Abstract base class for animals"""

    def __init__(self, name):
        self.name = name

    def speak(self):
        print("{} makes sound {}".format(self.name, self._voice()))

    def _voice(self) -> str:
        """Returns the sound made by this kind of animal"""
        raise NotImplementedError("Subclass must provide _voice method")

# Implement Sheep and Dog to get the behavior shown below.


class Sheep(Animal):

    def _voice(self):
        return "Baaaa"

class Dog(Animal):

    def _voice(self):
        return "Arf"


marigold = Sheep("Marigold")
charlie = Dog("Charlie")

marigold.speak()  # Expected output:  "Marigold makes sound Baaaa"
charlie.speak()   # Expected output:  "Charlie makes sound Arf"
```

Practice Midterm (Subclassing):

Question Type: What does this print?

```
class Shape(object):
    """Abstract base class for 2D shapes"""

    def area(self) -> float:
        """Each subclass must implement an 'area' method"""
        raise NotImplementedError("area method must be implemented")

    def bigger(self, other) -> bool:
        return self.area() > other.area()

class Rect(Shape):
    """A rectangle defined by lower left (ll) and upper right (ur) points"""

    def __init__(self, llx: float, lly: float, urx: float, ury: float) -> None:
        self.llx = llx
        self.lly = lly
        self.urx = urx
        self.ury = ury

    def area(self) -> float:
        return (self.urx - self.llx) * (self.ury - self.lly)


class Square(Rect):
    """A rectangle with equal length sides"""

    def __init__(self, llx: float, lly: float, side: float):
        self.llx = llx
        self.lly = lly
        self.urx = llx + side
        self.ury = lly + side

def q3():
    r = Rect(5.0, 5.0, 7.0, 7.0)
    s = Square(1.0, 1.0, 3.0)
    if r.bigger(s):
        print("Rectangles always win! Bigness of {}".format(r.area()))
    elif s.bigger(r):
        print("Squares rule! Bigness of {}".format(s.area()))
    else:
        print("Apples versus oranges! You just can't know.")
```

What does q3() print?

Squares rule! Bigness of 9.0

Listeners

Idiom: Hooks

```
class AbstractListener(object):
    """My subclasses respond to events"""
    ...
class AbstractListenable(object):
    """My subclasses generate events"""
    ....
class ConcreteListenable(AbstractListenable):
    """I do things and generate events"""
    ...
class ConcreteListener(AbstractListener):
    """I respond to events"""
    ...

spaceman = ConcreteListenable("Major Tom")
ground_control = ConcreteListener()
spaceman.add_listener(ground_control)
spaceman.do_something("floating in my tin can")
```

Connecting listener to listenable

```
spaceman = ConcreteListenable("Major Tom")
ground_control = ConcreteListener()
spaceman.add_listener(ground_control)
spaceman.do_something("floating in my tin can")
```

Output:

I am responding to 'I am Major Tom and I am floating in my tin can'

This example is based on Model-View-Controller, but "hooks" are more generally useful as a way of dynamically configuring objects. Example: Memory-mapped IO in the Duck Machine DM2018S is simulated with hooks.

The listeners

```
class AbstractListener(object):
    """My subclasses respond to events"""
    def notify(self, msg: str):
        raise NotImplementedError("Instantiate this is the subclass")

class ConcreteListener(AbstractListener):
        """I respond to events"""
        def notify(self, msg: str):
            print("I am responding to '{}'".format(msg))


ground_control = ConcreteListener()
spaceman.add_listener(ground_control)
spaceman.do_something("floating in my tin can")
```

The listenable

```
class AbstractListenable(object):
    """My subclasses generate events"""
    def __init__(self):
        self.listeners = [ ]

    def add_listener(self, listener: AbstractListener):
        self.listeners.append(listener)

    def notify_all(self, msg: str):
        for listener in self.listeners:
            listener.notify(msg)

class ConcreteListenable(AbstractListenable):
    """I do things and generate events"""
    def __init__(self, name: str):
        super().__init__()
        self.name = name

    def do_something(self, what: str):
        self.notify_all("I am {} and I am {}" .format(self.name, what))
```

Alyssa's Notes (Listeners)

<table>
<tr>
<td>Listener</td>
<td>Superclass:


```
class GeneralListener(object):
"""My subclasses will respond to events"""
```

1. Create a `def notify(self, msg):` method and this will probably have a `NonImplementationError` to indicate all of the listener subclasses need this method.

Subclass(es):

```
class SpecificListner(GeneralListener):
"""I respond to events"""
```

1. Create your `def notify(self, msg):` methods in each subclass.
    a. This will always have a print statement in it that is responding to something.
    b. This might increment something (such as incrementing the number of ingredients being added to your salad(salad=Listenable).</td>
</tr>
<tr>
<td>Listenable</td>
<td>Superclass:


```
class GeneralListenable(object):
""" My subclasses generate events"""
```

1. This is where you are going to have your constructor method
    a. This will definitely be where you initialize your `self.listeners = []` list
    b. You may also want to initialise another variable in your constructor, such as an empty list for ingredients to be added to.
2. You may (probably will) have a `def add_listener(self, listener: GeneralListener):` method, and this method will only be appended the listener to the list we just initialized: `self.listeners.append(listeners)`
3. You will create your `def notify_all(self, msg):` in this class! This method is going to be calling your notify method from your Listener class, and this method will be called whenever something happens.
    a. `def notify_all(self, msg):`</td>
</tr>
</table>

```
            for listener in self.listeners:
                listener.notify(msg)
```

Subclass(es):

```
class SpecificListenable(GeneralListenable):
"""I do things and generate events"""
```

1. This will have another `__init__` constructor method where it will inherit all of the initialisers in the superclass by using `super.().__init__()` and then it will also initialize its other parameters (such as this SpecificListenable's name... example = Salad)

2. Now is when you will actually have the method where something is being done, and this is when you will call upon the notify_all method that was created in the superclass.

    a.
```
def do_something(self, what):
    self.notify_all("I am {} and I am
        {}).format(self.name, what))
```

Examples from Piazza for LISTENERS:

```
"Let's make a listener"

class Pizza(object):

    def __init__(self):
        self.ingredients = [ ]
        self._listeners = [ ]

    def add_ingredient(self, ingredient: str):
        self.ingredients.append(ingredient)
        self.notify_all()

    def add_listener(self, listener):
        self._listeners.append(listener)

    def notify_all(self):
        for listener in self._listeners:
            listener.notify(self)

class Listener(object):

    def notify(self, pizza: Pizza):
        raise NotImplementedError("You forgot to define the notify method")

class I_Like_All_Pizza(Listener):
    def notify(self, pizza: Pizza):
        print("Yum! Pizza with {}".format(pizza.ingredients))

class I_am_picky(Listener):
    def notify(self, pizza: Pizza):
        if len(pizza.ingredients) <= 3:
            print("Pizza with {} sounds ok".format(pizza.ingredients))
        else:
            print("Too many ingredients!")

pizza = Pizza()
pizza.add_ingredient("crust")
pizza.add_listener(I_am_picky())
pizza.add_ingredient("sauce")
pizza.add_ingredient("cheese")
pizza.add_ingredient("pineapple")
```

Practice Exam Questions focused on understanding LISTENERS:

Midterm (Listeners):

Question Type: What does this print?

```
class Listener:
   def notify(self, subject: "Sandwich"):
       raise NotImplementedError("Sandwich listeners must implement notify")

class Sandwich(object):
    def __init__(self):
        self._listeners = [ ]
        self.ingredients = [ ]

    def add_listener(self, listener: Listener):
        self._listeners.append(listener)

    def _notify_all(self):
        for listener in self._listeners:
            listener.notify(self)

    def add(self, ingredients: list):
        for ingredient in ingredients:
            self.ingredients.append(ingredient)
        self._notify_all()

class Gluttony(Listener):
    def notify(self, sandwich: Sandwich):
        if len(sandwich.ingredients) > 4:
            print("{} is too many ingredients!".format(len(sandwich.ingredients)))

class Sloth(Listener):
    def notify(self, sandwich: Sandwich):
        if len(sandwich.ingredients) < 3:
            print("Better add something more")

my_sandwich = Sandwich()
my_sandwich.add_listener(Sloth())
my_sandwich.add_listener(Gluttony())
my_sandwich.add(["Bread", "Mayonnaise", "Mustard"])
my_sandwich.add(["Lettuce", "Chicken"])
```

What does this program print?

5 is too many ingredients!

Practice Midterm (Listener):

Question Type: Finish this method.

```
Point moved to 9,8
Point moved to 12,10
```

Finish the PointListener class so that the program produces that output. *This is a simplified version of the model-view-controller design we have used in several of our projects.*

```
class Point(object):
    """Point.x and Point.y are 'public' fields"""

    def __init__(self, x, y):
        self.listeners = [ ]
        self.x = x
        self.y = y

    def add_listener(self, listener: "PointListener") -> None:
        self.listeners.append(listener)

    def notify_all(self):
        for listener in self.listeners:
            listener.notify(self)

    def move(self, dx, dy):
        self.x += dx
        self.y += dy
        self.notify_all()


class PointListener(object):
    """This is the class you had to complete"""

    def __init__(self):
        """Probably not necessary"""
        pass

    def notify(self, pt: Point) -> None:
        """This is the method you needed to write.
        The expected output below tells you what
        the 'notify' method needs to print.
        """
        print("Point moved to {},{}".format(pt.x, pt.y))


p = Point(5,5)
p.add_listener(PointListener())
p.move(4,3)   # Expected output: "Point moved to 9,8"
p.move(3,2)   # Expected output: "Point moved to 12,10"
```

Practice Final (Listener):

Question Type: What does this print?

```
class Listener(object):
    def notify(self, event: dict):
        raise NotImplementedError("You most override notify")

class Listenable(object):
    def __init__(self):
        self.listeners = [ ]

    def add_listener(self, listener: Listener):
        self.listeners.append(listener)

    def notify_all(self, event: dict):
        for listener in self.listeners:
            listener.notify(event)

class SillyWalk(Listenable):
    def __init__(self, steps: list):
        super().__init__()
        self.steps = steps

    def walk(self):
        for step in self.steps:
            self.notify_all({ "step": step })

class StepWatcher(Listener):
    def __init__(self):
        self.count = 0

    def notify(self, event: dict):
        self.count += 1
        print("Step {}: {}".format(self.count, event["step"]))

silly = SillyWalk( ["kick", "wiggle", "swing"])
silly.add_listener(StepWatcher())
silly.walk()
```

What does it print?

Step 1: kick
Step 2: wiggle
Step 3: swing

This is a variation on the way we attach listeners to objects in the model-view-controller pattern. I took off 3 for missing the count, 3 for extraneous output at the end, 7 for treating them as being printed all together (there are three distinct calls to print).

Steps on how to attack: LISTENERS

1. Look at the superclass and the pre-written subclasses and what class they inherit upon. This will tell you exactly which class is a Listener and which is a Listenable.

2. How to tell if it is a Listener:

    a. This will be something that is responding to events (having a print statement telling you what is going on).
    Example: A reporter reporting on an Athlete running.

    b. This will be inheriting from the `GeneralListener` superclass.

    c. This will ALWAYS have a `notify` method. **

    d. Usually no constructor for Listeners.

3. How to tell if it is a Listenable:

    a. This will be something that is doing something and generating events.
    Example: An Athlete running.

    b. This will be inheriting from the `GeneralListenable` superclass.
        i. This superclass will have:
            1. constructed a `__init__` method that initializes an empty list of listeners.
            2. An `add_listener` method that will append to that empty list
            3. A `notify_all` method that is going to loop through the listeners and then call upon the notify method from the Listener class. **

    c. The subclass might have another `__init__` constructor inheriting from the superclass and initializes anything else it needs (such as it's name).

    d. The subclass Listenable will then have a method for the action it is actually doing and this method will call upon the `notify_all` method from the `GeneralListenable` superclass which calls upon the notify method from the Listener subclass.

Factoring Base Case

Factoring behavior into abstract base classes

```
class AbstractBase(object):

  def common_behavior(self, value: int):
    return 2 * self.specific_behavior(value)

  def specific_behavior(self, value: int):
    raise NotImplementedError("specific behavior needed in concrete classes")

class ConcreteSubClass(AbstractBase):

  def __init__(self, v: int):
    self.v = v

  def specific_behavior(self, value: int):
    return self.v * value

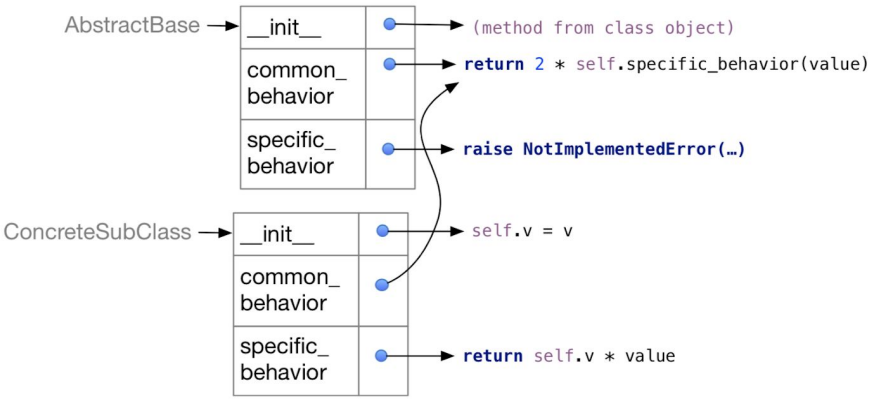
my_thing = ConcreteSubClass(10)
print(my_thing.common_behavior(5))
```

Idiom: Factored methods

```
class ConcreteSubClass(AbstractBase):

  def __init__(self, v: int):
    self.v = v

  def specific_behavior(self, value: int):
    return self.v * value
```



OOP Recursion / Algorithms:

-Loop design
-Recursion
-Plan composition

Alyssa's Notes (OOP Recursion / Algorithms)

- If (some subclass of) class C can contain objects of class C, then we have a recursive data structure.
- Recursive algorithms on recursive structures:
  - How to design a recursive function:
    - Identify the base case: compute a value directly
    - Identify the recursive case: combine result from recursive calls.

- Note: Base case and recursive case may be in methods of different subclasses!

```
class Shape(object):
     def area(self) -> int:
          raise NotImplementedError("subclass should implement
               'area' method")
class Rect(Shape):
     ''' ll = lower left, ur = upper right'''
     def __init__(self, p1: Point, p2: Point):
          self.ll = Point(min(p1.x, p2.x), min(p1.x, p2.y))
          self.ur = Point(max(p1.x, p2.x), max(p1.y, p2.y)
     def __repr__(self) -> str:
          return "Rect({}, {})".format(self.11, self.ur)
     def area(self) -> int: ** Base case! **
          return(self.ur.x - self.11.x) *
          (self.ur.y - self.11.ur)

class Composite(Shape):
     def __init__(self):
          self._parts = []
     def add(self, element: Shape):
          self._parts.append(element)
     def __repr__(self) -> str:
          return "Composite({})".format(self._parts)
```

... but now out Composite subclass of Shape needs an 'area' method. As I said before, if (some subclass of) class C can contain objects of class C, then we have a recursive data structure.
Now we need to build our own nested (recursive) structures.

Base Case: The shape. Compute the value directly.
Recursive Case: The multiple shapes added together. Combines results via recursive calls.

Note: The base case and recursive case methods can be in different subclasses!

```
    def area(self) -> int:   ** Recursive case! **
    '''Combined area of the parts'"
        result = 0
        for shape in self._parts:
            result += shape.area()  You can see here is where the area
        return result               would be needing to be adding the
                                    other shapes together from the Shape
                                    superclass and this combines the result
```

Some common algorithms on recursive structures:

- Find extreme element (max, min,...)
- Sum, product, etc. of elements
- Select elements with some property

These are all things you already know how to do with a list. We can do the same things with any recursive structure

Class example:

```
class Scene(object):
    def select(self, at_least, at_most):
        raise NotImplementedError("I am not here")
class Shape(Scene):
    def __init__(self, name: str, size: int):
        self.name = name
        self.size = size
    def __repr__(self):
        return "Shape({}, {})".format(self.mame, self.size)
    def select(self, at_least, at_most) -> List: ** Base Case! **
    ''' You want to see if the size of the shape is within the
    at_least and at_most parameters and return it as a list it is.'''
        if self.size > at_most or self.size < at_least:
            return []
        else:
            return [self]
class Composite(Scene):
    def __init__(self):
```

```
        self.parts = []  Need to initialize this list in the constructor
    def add(self, scene: Scene):
        self.parts.append(scene)
    def select(self, at_least, at_most) -> List:  *Recursive Case!*
        self.size_list = []
        for part in self.parts:
            result += part.select(at_least, at_most)
        return result

my_scene = Composite()
my_scene.add(Shape("oval", 12))
my_scee.add(Shape("square", 7))
subscene = Composite()
subscene.add(Shape("rect", 10))
my_scene.add(subscene)
print(my_scene.select(8, 15))
```

Practice Exam Questions focused on understanding OOP Recursion:

Midterm (OOP Recursion):

Question Type: Finish this method.

6. [10 points] Complete classes Basic and Composite so that that Food.calories() sums the calories of all ingredients.

```
class Food(object):
    """Abstract base class for foods, basic or composite."""

    def calories(self) -> int:
        """Should return total calories"""
        raise NotImplementedError("calories method not implemented")

class Basic(Food):
    """Not further decomposed into ingredients"""
    def __init__(self, name: str, calories: int):
        self.name = name
        self._calories = calories

    def calories(self) -> int:
        return self._calories

class Composite(Food):
    """Something we made from other ingredients"""
    def __init__(self, name: str, ingredients: list):
        self.name = name
        self._ingredients = ingredients

    def calories(self) -> int:
        total = 0
        for ingredient in self._ingredients:
            total += ingredient.calories()
        return total


salt = Basic("salt", 0)
crust = Composite("crust", [Basic("flour", 20), salt,  Basic("yeast", 3)])
tomatoes = Basic("tomatoes", 10)
sauce = Composite("sauce", [tomatoes, Basic("garlic", 2), salt])
cheese = Basic("mozzarella", 30)
margherita = Composite("pizza margaritta", [crust, sauce, cheese, tomatoes])
print("{} calories in pizza margherita".format(margherita.calories()))
# Expect:
# 75 calories in pizza margherita
```

Practice Midterm (OOP Recursion):

Question Type: Finish this method.

8. [10 points] The `max_leaf` method for the Tree structure below should return the maximum of leaf values. Complete the method. Be brief. (It may help to remember that the built-in function max(m,n) returns the maximum of its arguments.)

```
class Tree(object):
    """Abstract base class"""

    def max_leaf(self) -> int:
        raise NotImplementedError("max_leaf must be overridden")


class Leaf(Tree):
    """Leaves of tree hold positive integers"""

    def __init__(self, val: int) -> None:
        assert val > 0
        self.val = val

    def max_leaf(self) -> int:
        """You needed to write this basis case"""
        return self.val



class Interior(Tree):
    """Interior node has two subtrees"""

    def __init__(self, left: Tree, right: Tree) -> None:
        self.left = left
        self.right = right

    def max_leaf(self) -> int:
        """You needed to write this recursive case,
        or something equivalent in function.
        """
        return max(self.left.max_leaf(), self.right.max_leaf())




t = Interior(Leaf(4),Interior(Leaf(5),Leaf(3)))
assert t.max_leaf() == 5
```

Final Practice Midterm (OOP Recursion):

Question Type: Finish this method.

5. [10 points] Finish the `population` methods so that `Area.population` returns the sum of populations of all the `Places` in the area. See below for an example that sums the populations of Eugene, Springfield, Corvallis, and Philomath.

```
from typing import List

class Area(object):
    """Abstract class."""
    def __init__(self, name: str):
        self.name = name

    def population(self) -> int:
        raise NotImplementedError("population method nust be overridden")


class Place(Area):
    def __init__(self, name: str, pop: int):
        self.pop = pop
        super().__init__(name)

    def population(self) -> int:
        return self.pop


class Region(Area):
    def __init__(self, name, subregions: List[Area]):
        super().__init__(name)
        self.subregions = subregions

    def population(self) -> int:
        pop = 0
        for subregion in self.subregions:
            pop += subregion.population()
        return pop

lane = Region("Lane County", [Place("Eugene", 166575), Place("Springfield", 60177)])
benton = Region("Benton County", [Place("Corvallis", 55298), Place("Philomath", 4594)])
s_willamette = Region("South Willamette Valley", [lane, benton])
print(s_willamette.population())   #Expected output: 286644
```

Almost everyone got the base case right. A lot of people had trouble with the inductive case, often trying to access fields of the elements of `self.subregions` instead of just making the recursive method call. Some made the call but failed to add up the results, which puzzles me ... summing elements in a list is another very common idiom that you should be able to do in your sleep. I guess that you have more trouble with these common algorithms when they are set in the context of something else, like the recursive traversal.

Steps on how to attack: OOP RECURSION / Algorithms

1) Figure out the Base Case.
    - This will be something that computes the value directly. It does not call upon anything else, and it it's own, straight away value.
    - Ex 1) The area of a rectangle is (RectA = l * w) and that is the only way to do it, and you would only need to find the area of one Rect at a time.
    - Ex 2) The calories of bread would only be the calories of that bread, this does not matter upon anything else.

2) Figure out the Recursive Case.
    - This will be needing to combine the result with others from a recursive call. This would need to be adding values together (maybe/likely from different subclasses).
    - Ex 1) You can add the areas of the shapes together as a composite shape. This would be like needing to create a house and needing to add
      RectangeArea + TriangleArea + SquareArea = HouseArea
    - Ex 2) Finding the amount of calories in a sandwich would be
      BreadCalorie + MeatCalorie + VeggieCalorie + SpreadCalorie = SandwhichCalorie

How to know when to do OOP Recursion:

- If (some subclass of) class C can contain objects of class C, then we have a recursive data structure.
    - Ex) A composite area of shapes would need to contain each shape object to calculate the total area.

-- EXAMPLE --

```
class Shape: ...
class Rectangle: ...
        def area(self) -> int: ** Base case! **
                return(self.ur.x - self.11.x) * (self.ur.y - self.11.ur)
class Composite:
        def add(self, element: Shape):
                self._parts.append(element)  ** Recursive case! **
```

-- EXAMPLE --

```
class Food: ...
class Basic(Food):
        def __init__(self, name, calories):
                self.name = name
                self._calories = calories

        def calories(self) -> int:
                return self._calories   ** Base case! **

class Composite(Food):
        def __init__(self, name, ingredients):
                self.name = name
                self._ingredients = ingredients
        def calories(self) -> int:
                total = 0
                for ingredient in self._ingredients:
                        total += ingredient.calories()   ** Recursive case! **
                return total
```

Two-Step Algorithms:

Alyssa's Notes (Two-Step Algorithms)

(Look at Assembler Pass 1)
Pass 1: Look through all the data
Pass 2: Use that data we looked through

Do I need to go through everything to know what to do next? (i.e. going through a list of all of the integers and then find the max number after we go through all that).

Class / Midterm Example:

Grid:

```
[ [ 1,  2,  3],    row
  [ 4,  2,  1],
  [ 1,  3,  2]]
 = 6   7   6
```

=> [[1, 4, 1], [3, 1, 2]]

```
columns = []
for i in range(len(grid[0])):
     col = []
     for row in grid:
          col.append(row[i])
     columns.append(col)
```

Columns...

```
target = 0
selected = []
for col in columns:
     if sum(col) == target:
          selected.append(col)
return selected
```

Practice Exam Questions focused on understanding TWO STEP ALGORITHMS:

Midterm (Two-Step Algorithm):

Question Type: Finish this method.

5. [10 points]

We will say a square grid of integers is *balanced* if and only if the sum of each row and the sum of each column is the same. For example, the grid at the right is balanced because each row and each column sums to 15. Complete the `is_balanced` method in class NumberGrid to check this property.

2	7	6
9	5	1
4	3	8

```
from typing import List

class NumberGrid(object):
    """A square grid of numbers"""
    def __init__(self, values: List[List[int]]):
        self.values = values
        for row in values:
            assert len(row) == len(values)  # Grid is square

    def is_balanced(self) -> bool:
        """Balanced if sum of each row and column is equal"""
        target = 0
        for item in self.values[0]:
            target += item
        # Each row the same
        for row in self.values:
            total = 0
            for item in row:
                total += item
            if total != target:
                return False
        # Each column the same
        for col in range(self.size):
            total = 0
            for row in range(self.size):
                total += self.values[row][col]
            if total != target:
                return False
        return True


assert NumberGrid([[2,  7, 6], [9, 5, 1], [4, 3, 8]]).is_balanced()
assert not (NumberGrid([[2,  7, 6], [9, 7, 1], [4, 3, 8]]).is_balanced())
print("Passed two simple test cases on 3x3 grid")
```

Practice Final (Two-Step Algorithm):

6. [10 points] Sometimes a project is really hard. For example, it might be worth 100 points, but maybe the highest score anyone earns is 42. In such a case I might want to use a *normalized* score based on the highest actual score. For example, the student whose score is 42 (the highest in the class) would receive a normalized score of 100, and a student who earned 21 would get a normalized score of 50, because 21 is half of 42.

Finish the method "normalize" below to compute adjusted scores. You can round down to an integer, or not, as you prefer.

```
from typing import List

class Score(object):
    def __init__(self, name: str, score: int):
        self.name = name    # Public, read-only
        self.score = score  # Public, read-only

    def __repr__(self):
        return 'Score("{}",{})'.format(self.name, self.score)

def normalize(projects: List[Score]) -> List[Score]:
    """ Returns list of scores normalized as percent
    of max score.
    """
    if len(projects) == 0:
        return projects
    max = projects[0].score
    for project in projects:
        if project.score > max:
            max = project.score
    result = [ ]
    for project in projects:
        result.append(Score(project.name, int(100 * project.score / max)))
    return result


scores = [Score("Leslie", 40), Score("Bobby", 23), Score("Adrian", 42)]
print(normalize(scores))
# Expected output: [Score("Leslie",95), Score("Bobby",54), Score("Adrian",100)]
```

This problem is about two-pass algorithms, of course, and I gave at least 5 points if you had the basic two pass strategy (one pass to get the maximum, a second pass to normalize the scores). I knocked off one point if you modified the input list; I believe we've talked in class about the expectation that a function that returns something does not modify its inputs. A surprising number of you had trouble with accessing fields of the Score objects; typically you lost 4 points for that. Many returned a list of strings instead of a list of Score objects (violating the function header type declarations), for which I deducted 3. There were many mathematical mistakes in the calculation (e.g., doing an integer division before scaling by 100), but I didn't deduct for them.

Examples from Piazza for TWO-STEP ALGORITHMS:

Two-pass algorithm from lecture

Here's the code we wrote:

```
"""A problem for which you need a two pass algorithm"""

from typing import List

def label(fruits: List[str]) -> List[str]:
    fruit_labels = []
    # pass 1:  How many of each fruit?
    counts = { }
    for fruit in fruits:
        if fruit in counts:
            counts[fruit] += 1
        else:
            counts[fruit] = 1
    # Pass 2: Label each fruit
    count_current = { }
    for fruit in fruits:
        if fruit in count_current:
            count_current[fruit] += 1
        else:
            count_current[fruit] = 1
        fruit_labels.append("{} {} of {}".format(
            fruit,count_current[fruit], counts[fruit] ))

    return fruit_labels


fruits = ["apple", "banana", "banana", "berry", "apple",  "banana"]

print(label(fruits))

# Output should be:
#  ['apple 1 of 2', 'banana 1 of 3', 'banana 2 of 3', 'berry 1 of 1', 'apple 2 of 2', 'b
anana 3 of 3']
```

Steps on how to attack: TWO-STEP ALGORITHMS

How to know if you should do a Two-Step Algorithm...

- Do I need to go through everything to know what to do next?
  (ex: going through a list of all of the integers and then find the max number after we go through all that).

The structure of a Two-Step Algorithm:

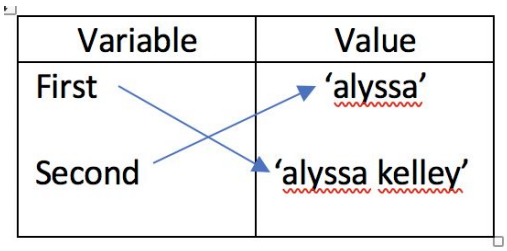
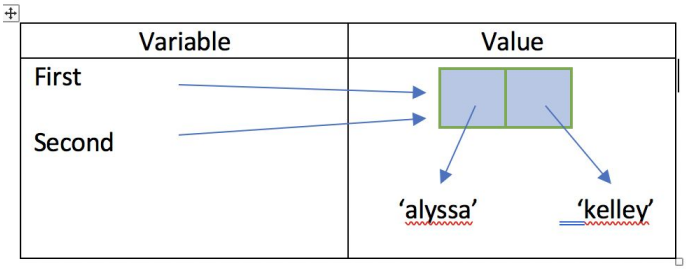
- Pass 1: Look through all the data
- Pass 2: Use that data we looked through to determine something.

Memory Model / Aliasing:

Alyssa's Notes (Aliasing)

- An alias is a second name for a piece of data and it is often easier and more useful to create a second copy. If the data is immutable (which means it cannot be modified in space aka Tuples: `name = ('Alyssa', 'Kelley'))` then aliasing does not matter because the data cannot change no matter how many times it is referred to. But if the data is mutable (which means it can change in place like lists: `line = [] / line = line`.append('hello')) and this makes it hard to find bugs.
- Aliasing happens whenever one variable's value is assigned to another variable

Immutable	Mutable

`name = 'alyssa'`
`second = first`
 	
`first = first + 'kelley'`
 	
 
 	
`first` now is pointing to `'alyssa kelley'` and second is still pointing to `'alyssa'` and this doesn't mess anything up. Whenever we change a string, python creates a new string behind the scene.
	`first = ['alyssa']`
	`second = first`
	`first = first.append('kelley')`
	`print first`
	`['alyssa', 'kelley']`
	`print second`
	`['alyssa', 'kelley']`
	
	 
	
	Lists can be changed in place. We have two variables referring to the same memory and so when something changed (append) then it changed both because technically both are the same value. We didn't want second to change, but it did as a side effect.

Steps on how to attack: Aliasing

- Draw a diagram and thinking about if more than one variable is using the same memory location. Draw each object and arrows to that object as I did in the chart above.

Practice Exam Questions focused on understanding ALIASING:

Midterm (Aliasing):

Question Type: What does this print? / Find the error.

```
class Point(object):
    def __init__(self, x: int, y: int):
        self.x = x
        self.y = y

    def move(self, dx: int, dy: int):
        self.x += dx
        self.y += dy

    def __str__(self):
        return "Point({}, {})".format(self.x, self.y)

class Rect(object):
    def __init__(self, ll: Point, width: int, height: int):
        self.ll = ll
        self.height = height
        self.width = width

    def move(self, dx: int, dy: int):
        self.ll.move(dx, dy)

    def ur(self) -> Point:
        return Point(self.ll.x + self.width, self.ll.y + self.height)

    def __str__(self):
        return "Rect({}, {})".format(self.ll, self.ur())

corner = Point(10,10)
small_rect = Rect(corner, 10, 10)
larger_rect = Rect(corner, 20, 20)
larger_rect.move(20,20)
print("Smaller rect is {}".format(small_rect))
print("Larger rect is {}".format(larger_rect))
```

The programmer was hoping it would print

```
Smaller rect is Rect(Point(10, 10), Point(20, 20))
Larger rect is Rect(Point(30, 30), Point(50, 50))
```

But it doesn't! What does it actually print?

Smaller rect is Rect(Point(30, 30), Point(40, 40))
Larger rect is Rect(Point(30, 30), Point(50, 50))

Regular Expressions:

https://docs.python.org/3/howto/regex.html

Regular Expression Basics	
.	Any character except newline
a	The character a
ab	The string ab
a|b	a or b
a*	0 or more a's
\	Escapes a special character

Regular Expression Quantifiers	
*	0 or more
+	1 or more
?	0 or 1
{2}	Exactly 2
{2, 5}	Between 2 and 5
{2,}	2 or more
{,5}	Up to 5

Default is greedy. Append ? for reluctant.

Regular Expression Groups	
(...)	Capturing group
(?P<Y>...)	Capturing group named Y
(?:...)	Non-capturing group
\Y	Match the Y'th captured group
(?P=Y)	Match the named group Y
(?#...)	Comment

Regular Expression Character Classes	
[ab-d]	One character of: a, b, c, d
[^ab-d]	One character except: a, b, c, d
[\b]	Backspace character
\d	One digit
\D	One non-digit
\s	One whitespace
\S	One non-whitespace
\w	One word character
\W	One non-word character

Regular Expression Assertions	
^	Start of string
\A	Start of string, ignores m flag
$	End of string
\Z	End of string, ignores m flag
\b	Word boundary
\B	Non-word boundary
(?=...)	Positive lookahead
(?!...)	Negative lookahead
(?<=...)	Positive lookbehind
(?<!...)	Negative lookbehind
(?()|)	Conditional

Regular Expression Flags	
i	Ignore case
m	^ and $ match start and end of line
s	. matches newline as well
x	Allow spaces and comments
L	Locale character classes
u	Unicode character classes
(?iLmsux)	Set flags within regex

Regular Expression Special Characters	
\n	Newline
\r	Carriage return
\t	Tab
\YYY	Octal character YYY
\xYY	Hexadecimal character YY

Regular Expression Replacement	
\g<0>	Insert entire match
\g<Y>	Insert match Y (name or number)
\Y	Insert group numbered Y

Alyssa's Notes (Regular Expression)

EXAMPLES:

abc	matches "abc"
x|y	matches x or y
	(yabba) | (dabba) matches yabba or dabba
	(y|d)abba also matches yabba or dabba
[a-zXY]	matches any character in 'a'- 'z' or 'X' or 'Y'
(boom)*	matches "", "boom", "boomboom", etc.
	(x* is zero or more x)
(boom)(boom)*	(or: (boom)+)

Examples:

Desired: "Great!", "grreat!", "Grrreat!", etc BUT NOT "Grat!" or "greaaat!"

How to achieve this:

[Gg]rr*eat!
(G|g)r+eat!
(Grr*)|(gr+)eat! etc...

DESCRIPTIONS:

. Matches any character except a newline

\ Used to escape special characters (to get a literal match on "*", for example), or to indicate the start of a special sequence (which will be '\' followed by an ASCII letter or digit).

* Causes the resulting RE to match 0 or more repetitions of the preceding RE, as many repetitions as are possible. ab* will match â€˜aâ€™, â€˜abâ€™, or â€˜aâ€™ followed by any number of â€˜bâ€™s.

+ Causes the resulting RE to match 1 or more repetitions of the preceding RE. i.e. 'ab', 'ab....b', NOT 'a'.

? Causes the resulting RE to match 0 or 1 repetitions of the preceding RE. ab? will match either â€˜aâ€™ or â€˜abâ€™.

[ ] used to indicate a set of characters, which can be specified in several ways.

?P<...> specifies a dictionary label that will act as a key to the corresponding matched field in the regex. Used with groupdict() method.

\s Matches all unicode whitespace characters

\w any character that can be in a word in most languages!

\d digit [0 - 9 and others]

\# Escapes the comment character- meaning it's a literal '#'

$ Matches only the END of the string. i.e. foo$ will match foo but not foobar.

METHODS:
regexobject.search(word) searches the whole word for any subsequence that matches the regex

regexobject.match(word) matches word from the beginning (but allows trailing characters that don't match)

regexobject.fullmatch(word) has to match the full word exactly.

EXAMPLE) Writing a REGEX that will match any phone number format
(541) 333 -3333 / 541-111-1111 / 541.111.1111 / 5411111111 etc..

```
PHONE_PAT = re.compile(r'''
```
you need this so python does not think that what we are entering is special characters like \n is now a \ and a n instead of a new line character.

`\( ?` \ = escape character so it ignores fancy meanings and turns it into literal so this is actually a parenthesis that would be used in the format of the phone number (541) and not just use to group together 541 inside a parenthesis
? = having that parenthesis at the beginning of the phone number is optional

`[0-9]{3}` [0-9] means any numbers between 0-9 and + mean this can be one or more of these numbers, and {3} specifies an exact number of used numbers from that 0-9 so there will be 3 of these [0-9] numbers

`\)?` ending the parenthesis from before and this is optional
`[\s.-]?` might see a space (s) a dot (.) or a dash (-) and this is all optional (?)
`[0-9]{4}` there will then be 4 more digits ({4}) that are within [0-9]
`''', re.VERBOSE)` this allows us to use whites paces without python getting mad

```
test_words = ['test', '(541) 333-2345', 3456789999, '541-222-3333']
regexes = [(PHONE_PAT, "PHONE_PAT")]
def test_regex():
    for regex in regexes:
        print("regex {} matches: ".format(regex[1])"
        for word in test_words:
            match = regex[0].fullmatch(word)]   * search / match / fullmatch
            if match is not None:
                print(word)
        print()
test_regex()
```

Professor's Example: (URL)

```
URL_PAT = re.compile(r"""
http(s)?://              # Introduce http or https protocol
([-a-zA-Z0-9]+\.)+       # Qualified domain, up to top-level domain
(?P<tld> [a-zA-Z0-9]+)   # Top-level domain
[/\?\"\' ]               # Something after URL
""", re.VERBOSE)
```

Steps on how to attack : Regular Expressions

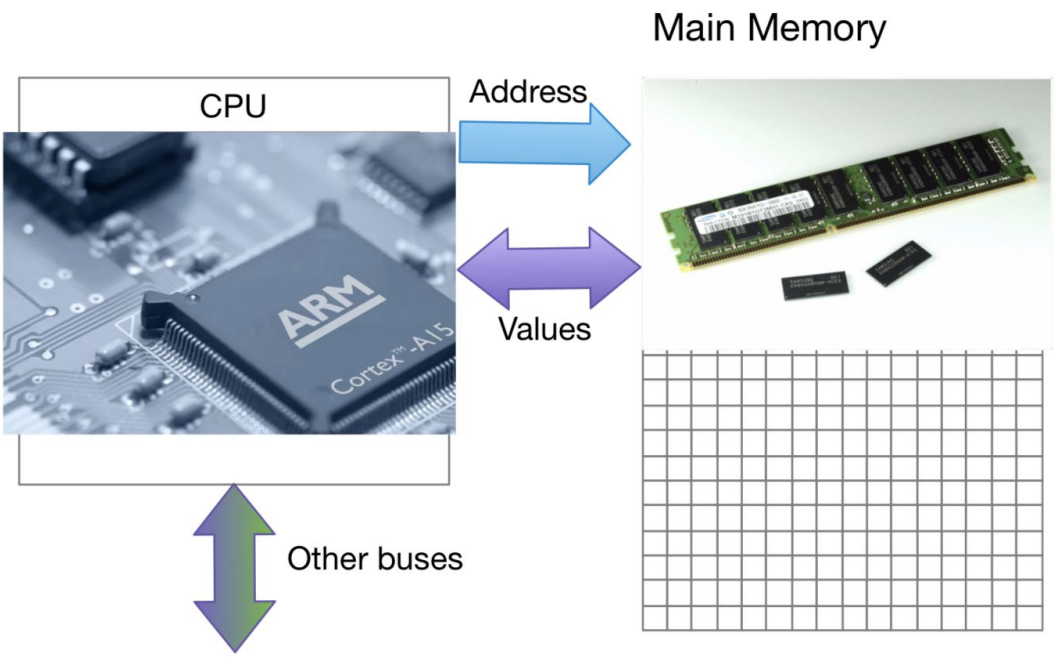
1. Create the pattern that you are needing to use.
2. Look at it step by step on what could happen next and make sure you think about if it could occur or it is will always occur.
3. add `(r'''` right after you create your pattern so you can include raw text and then before you use some symbol (such as wanting to include a (but not wanting it to be to group things off) then you need to include the front slash \ so python lets you
4. Look at the chart to see exactly how to do things and then what it is meaning
5. End it off with re.VERBOSE because that is how python lets you have whitespaces all in your pattern and not get bad at the formatting of it all.
6. You can then write a simple function that goes through each regular expression in your list of regular expressions and then through each word in your list of words to match it to and will see if it match part of your regular expression (there is the option of search, match, and a fullmatch--fullmatch is often used because it checks to see if it is fully matched) and then you can print the matched word. You can also add format line to have the word show that is matched the pattern.

Masking / Bit Manipulation:

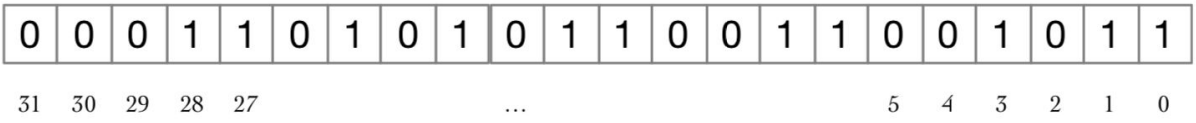
Alyssa's Notes (Masking / Bit Manipulation)

- Computer memory is binary
  - Everything (instructions, numbers, string...memory) is just one big array of binary numbers

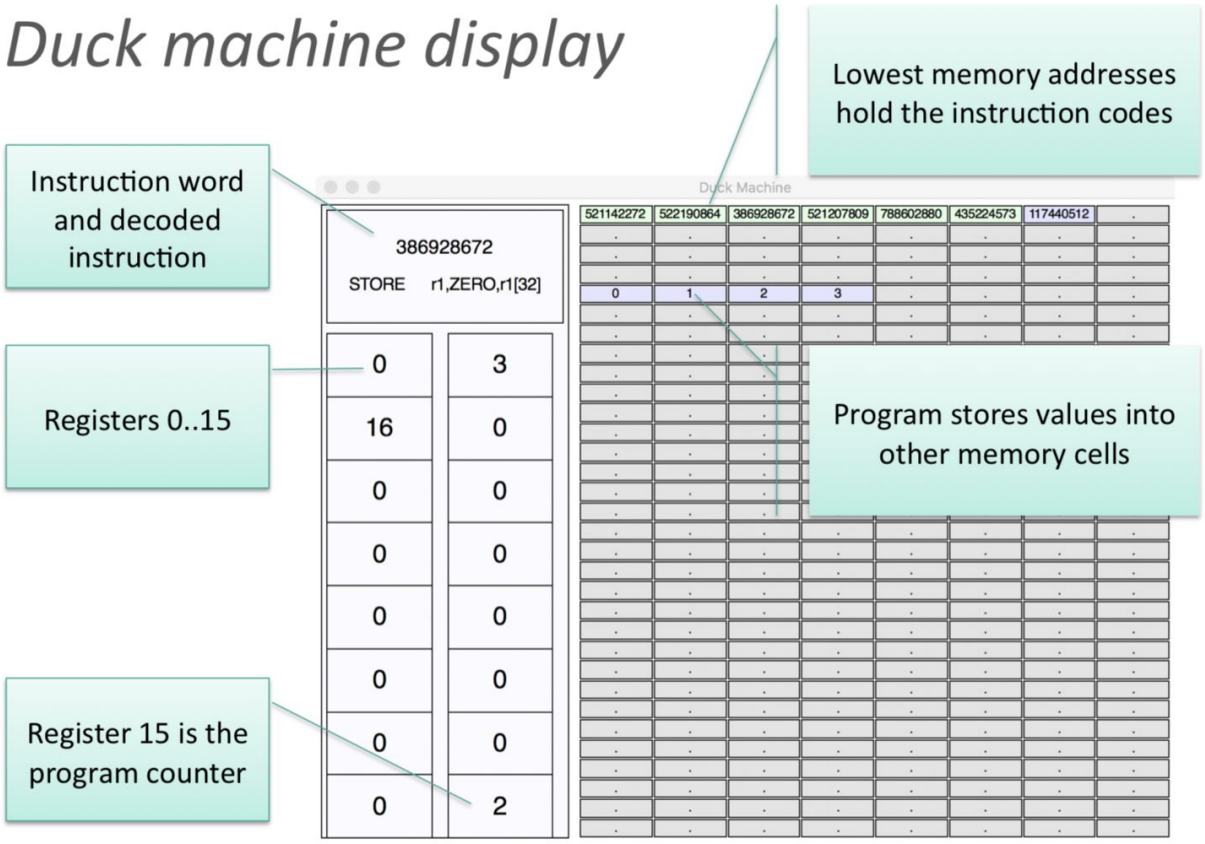
- 0 = False = 0 volts 1 = True = 5 volts



- A bit is a single binary digit
- A byte is a 8 binary digits
  - Most computer memory is "byte addressed"; a byte is the smallest addressable unit
  - A half byte is 4 bits and is called a nibble
- A "word" is a sequence of bytes
  - Usually 4 bytes = 32 bits



- Bit n has value 2^n
- 32-bit architecture:
  - 32-bit word addressed memory
  - 16 x 32-bit registers
    - Special registers:
    - r0 always holds zero
    - r15 is the program counter



We write DM2018S "assembly language" like this:
ADD/N r15, r0, r15[-3]
An "assembler" (assembly language translator) converts this instruction to:
435224573 (0x19f0fffd)

- PACKING (aka INSERTING a field into a word):
  Ex) word: ALYSSA / from_bit = 1 / to_bit = 2 / field = ZZ
  result = ALYZZA

```
mask = 0
for bit in range(field_weidth):
      mask = (mask << 1) + 1
return mask
```

This starts it off at 0 and then with shift over then 1 is added to it. This now creates a mask of all 1's and this is made to keep all of the 1's and 0's that were in the original bitfield.

You then need to create the mask that will 0-out that field and to do this you need to negate that mask that is created. To do this to be the specific length / area that you need, you have to take the mask of all 1's and then shift it to the left by the from_bit.

```
eraser = ~ (self.mask << self.from_bit)
         ~ (1 1 1 1     << location #      )
new_word_masked = eraser & word                          word 0000 word
new_field_word = field & self.mask
new_field_word = new_field_word << self.from_bit         0000 field 0000
```

Then we are creating the new word that has the new field inserted into it by taking the word that has the zeroed out field like this word0000word and we can OR it with the masked field 0000field0000 and this would allow all of the 1's in any location to be dropped down since it is OR.

```
new_word_inserted = new_word_mask | new_field_word
```

field word field

Note: there are 16 total registers (0-15).

The Operators:

x << y

Returns x with the bits shifted to the left by y places (and new bits on the right-hand-side are zeros). This is the same as multiplying x by 2**y.

x >> y

Returns x with the bits shifted to the right by y places. This is the same as //'ing x by 2**y.

x & y

Does a "bitwise and". Each bit of the output is 1 if the corresponding bit of x AND of y is 1, otherwise it's 0.

x | y

Does a "bitwise or". Each bit of the output is 0 if the corresponding bit of x AND of y is 0, otherwise it's 1.

~ x

Returns the complement of x - the number you get by switching each 1 for a 0 and each 0 for a 1. This is the same as -x - 1.

- Two's Complement (binary for negative integers):
  - ~(x-1)

- UNPACKING (aka EXTRACTING a field from a word)
  Ex) word: ALYSSA / from_bit = 1 / to_bit = 2
  field_extracted = SS

This would be taking a word and then returning the field. We first take the word and shift it to the RIGHT by the from_bit amount

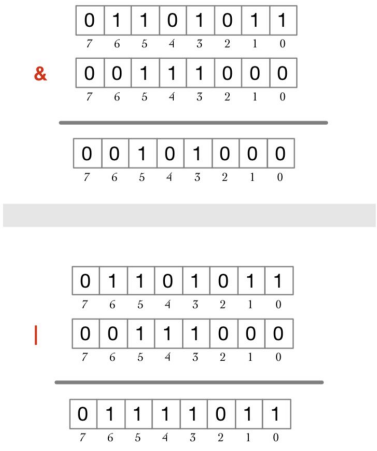
```
shifted_word = word >> self.from_bit
```

Shifts it over so the field is at the end starting at 0.

Once it is shifted all the way to the from_bit, then that means there are 0's on the left side of the field and then the field is still consisting of the original 1's and 0's which is what we want.
You can the AND this shifted_word with self.mask (which is all 1's) that is the length of the field_width and then return that extracted field

```
extracted_field = shifted_word & self.mask
return extracted_field
```

puting 1's where the field is so the field is dropping down and the rest is just zeros



Class Example:

7 4 3 0

--- X --- ---------- Y ---------- ---------- Z ----------

How to get a mask that is not creating a function, but using numbers...

32 16 8 4 2 1

2^5 2^4 2^3 2^2 2^1 2^0

So if you want a mask that is like 1 1 1 then you now you would need to add 4 + 2 + 1 = 7
So then you would have mask = 7 so it would be 1 1 1

1 = 2^0 = 1
11 = 2^0 + 2^1 = 2 + 1 = 3
111 = 2^0 + 2^1 + 2^2 = 1 + 2 + 4 = 7
1111 = 2^0 + 2^1 + 2^2 + 2^3 = 1 + 2 + 4 + 8 = 15

More on bit manipulation:
https://www.hackerearth.com/practice/notes/bit-manipulation/

Some hex / bin / decimal values to know by heart:

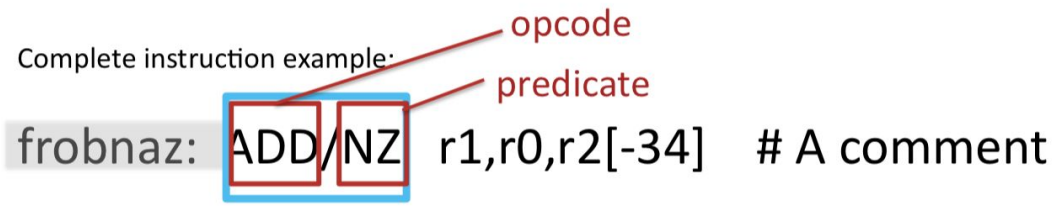
Hex	Binary	Decimal
0	0000	0
1	0001	1
7	0111	7
F	1111	15
FF	1111,1111	255

Assembly / Compiler

```
ASSM_PAT = re.compile(r"""
  (
   (?P<label> [a-zA-Z]\w*):
  )?   \s*
  ( (?P<opcode>    [a-zA-Z]+)          # Opcode
    (\/ (?P<predicate> [a-zA-Z]+) )?   # Predicate (optional)
    \s+
    (?P<target>    r[0-9]+),          # Target register
    (?P<src1>      r[0-9]+),          # Source register 1
    (?P<src2>      r[0-9]+)           # Source register 2
    (\[ (?P<offset>[-]?[0-9]+) \])?     # Offset (optional)
  )?
  (  \s*  (?P<comment>[\#;].*)  )?
  \s*$
  """, re.X)
```

It's just a sequence of simpler patterns, one for each part of a line of code, grouped into three optional blocks.

Complete instruction example:



Pattern fragment:

```
(?P<opcode>    [a-zA-Z]+)              # Opcode
   (\/ (?P<predicate> [a-zA-Z]+) )?   # Predicate (optional)
```

Steps on how to attack:

Make a picture that has registers in it LOAD R1 (draw R1 as a circle and put the number in it) and then continue to the next instruction.

Assembly language:

Count10 ASM

```
# Lovingly crafted by robots
 LOAD r1,const0_2
 STORE  r1,x_1
loop_3:  #While loop
 LOAD r1,x_1
 LOAD r2,const11_5
 SUB  r1,r1,r2
 SUB  r0,r1,r0
 JUMP/Z endloop_4
 LOAD r1,x_1
STORE r1,r0,r0[511] # Print
 LOAD r1,x_1
 LOAD r2,const1_6
 ADD  r1,r1,r2
 STORE  r1,x_1
 JUMP loop_3
endloop_4:
 HALT  r0,r0,r0
x_1: DATA 0 #x
const0_2:  DATA 0
const11_5:  DATA 11
const1_6:  DATA 1
```

Count10 DASM

```
c# Lovingly crafted by robots
LOAD r1,r0,r15[16]  # Access variable 'const0_2'
STORE r1,r0,r15[14]  # Access variable 'x_1'
loop_3:  #While loop
LOAD r1,r0,r15[13]  # Access variable 'x_1'
LOAD r2,r0,r15[14]  # Access variable 'const11_5'
 SUB  r1,r1,r2
 SUB  r0,r1,r0
 ADD/Z  r15,r0,r15[8] #Jump to endloop_4
LOAD r1,r0,r15[8]  # Access variable 'x_1'
STORE r1,r0,r0[511] # Print
LOAD r1,r0,r15[6]  # Access variable 'x_1'
LOAD r2,r0,r15[8]  # Access variable 'const1_6'
 ADD  r1,r1,r2
STORE r1,r0,r15[3]  # Access variable 'x_1'
 ADD  r15,r0,r15[-11] #Jump to loop_3
endloop_4:
 HALT  r0,r0,r0
x_1: DATA 0 #x
const0_2:  DATA 0
const11_5:  DATA 11
const1_6:  DATA 1
```

Practice Exam Questions focused on understanding MASKING:

Practice Final (Masking):

Question Type: Finish this method.

3. [10 points] Many early personal computers including the TRS-80 and Kaypro II used the Zilog Z80 8-bit microprocessor. The 8-bit instruction code for a Z80 is divided into three parts:

- x: a 2-bit field (bits 6..7)
- y: a 3-bit field (bits 3..5)
- z: a 3-bit field (bits 0..2)

In the program below I have provided a function "pack" for packing fields x, y, and z into an integer. Complete the corresponding function "unpack" for extracting the x, y, and z fields, such that unpack(pack$(x, y, z)) = x, y, z$ provided $0 \leq x \leq 3$, $0 \leq y \leq 7$, $0 \leq z \leq 7$.

```
from typing import Tuple

def pack(a: int, b: int, c: int) -> int:
    word = (a & 3) << 6 | (b & 7) << 3 | (c & 7)
    return word

def unpack(word: int) -> Tuple[int, int, int]:
    c = word & 7
    b = (word >> 3) & 7
    a = (word >> 6) & 7
    return a, b, c


w1 = pack(3,7,7)
assert unpack(w1) == (3,7,7)

w2 = pack(1,1,1)
assert unpack(w2) == (1,1,1)
```

I wouldn't ask you to remember the symbols & and |, but in this case they are all available to you in the pack function, so you didn't have to recall them if you could read and understand the code.

There are many ways to get this problem right and many ways to get it wrong. I took off 3 for using or (|) in place of and (&) for masking, 5 off for logical sequencing errors (e.g., basing the second value on the first value), 3 for shifting the wrong direction, 3 for masking before shifting (with the mask in the low bits), 1 for getting the field positon or width wrong (e.g., using a 2-bit mask when a 3-bit mask is needed).

Steps on how to attack: MASKING / BIT MANIPULATION

1. This about what section you want to KEEP and what section you want to REMOVE.
    a. KEEP: you need this to be a string of 1's
        i. Do this by creating a mask and shifting it << and adding 1 each time it is shifted. This will give you the 1111 you want.

    mask = 1111

    b. REMOVE: you need this to be a string of 0's
        i. Do this by taking the mask you created and negating it. You will need to shift this mask to the location that you are wanting to be removed so it will look like:

        ```
        eraser = ~ (self.mask << self.from_bit)
        ```

    masked_out = 0000
1. Now think about what you want to do when the fields are kept / removed. If you want to essentially ADD them together to create a joined word with both fields, then you would want to AND them.
2. If you want it to be only a certain word return that is the same from two different fields you would OR them together.

1010101	1010101
1110111	1110111
& (and)	| (or)
---	---
1010101	1110111

Some important numbers to remember when needing a mask:

1 = 1 = (2^0)
11 = 3 = (2^0 + 2^1)
111 = 7 = (2^0 + 2^1 + 2^2)
1111 = 15 = (2^0 + 2^1 + 2^2 + 2^3)

	32	16	8	4	2	1
2^5	2^4	2^3	2^2	2^1	2^0	
	5	4	3	2	1	0

So if you want a mask that is like 1 1 1 then you now you would need to add 4 + 2 + 1 = 7
So then you would have mask = 7 so it would be 1 1 1.

RandomTree Challenge: final practice

Here's another, slightly harder one:

CHALLENGE

write a class RandomTree

- it doesn't inherit from anything

- a RandomTree has several attributes:

- root : Node

- height : int

- max_children : int

- max_value : int

- write the __init__(self, height, max_children, max_value):

- __init__ should call the private method _build_tree(self).

_build_tree should construct the tree by instantiating and assigning a root node, and then recursively building from there.

The value of each node should be a random value between 0 and self.max_value.

The number of children for each node should be a random value between 0 and self.max_children.

The final height should be self.height.

- hint: what should the function signature be for _build_tree?

- do we need to pass in self.max_children? self.max_value?

- what about self.height?....

- write methods count and dfs for the tree

hint: these should be really easy!

(if you've already written your Node class)

Build some random-ish trees!

RANDOMTREE ANSWER

```
from node import Node
import random
import turtle as t

class RandomTree:

    def __init__(self, max_val, max_children, height):
        self.max_val = max_val
        self.max_children = max_children
        self.height = height
        self.root = Node(random.randint(0, self.max_val))
        self._build_tree(height, self.root)


    def _build_tree(self, height, root):
        if height == 0: # base case - stop recursing when tree is tall enough
            return
        num_children = random.randint(0, self.max_children)
        for i in range(num_children):
            root.add_child(Node(random.randint(0, self.max_val)))
        for child in root.children: # recursively build subtrees on each child
            self._build_tree(height - 1, child)

    def count(self):
        return self.root.count()

    def dfs(self, target):
        return self.root.dfs(target)

** this is where the turtle graphic functions would go **

tree = RandomTree(9, 3, 4)
print("Generated {} nodes".format(tree.count()))
tree.draw_tree()
```

Node challenge: final practice

Here's the spec for a challenge problem given out in lab. Will post solutions next week sometime.

Note, this is NOT based on anything we know about the final exam (because we know nothing), it's just a good exercise.

Write a class Node

- it doesn't inherit from anything

- a Node has two attributes:
  - value : Real
  - children : List[Node]
- write the following special methods:
  - __init__(self, value, children)
  - __str__(self)
  - __repr__(self)
  - __eq__(self, other)
  - __iter__(self) # we want to iterate on children
- write the following non-special methods:

  Hint: two are recursive
  - add_child(self, child: Node):

    '''adds Node child to children attribute'''
  - count(self):

    '''return a count of the total number of nodes,

    including this one, in the subtree rooted at this node.'''

  - dfs(self, target: Real):

    '''return True if any node in the subtree rooted at this node contains

    the value "target", False otherwise.'''

    ### dfs stands for "depth-first search"

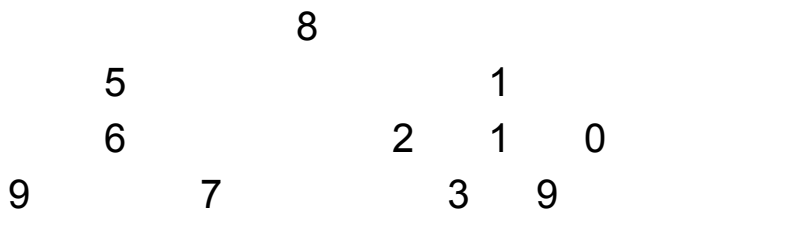
Write a subclass Leaf

- Leaf inherits from Node
- it is almost the same, but it doesn't have any children.
- it provides the base cases for our recursive count and dfs methods

When your classes are written...:

- instantiate a variety of nodes / leaves
- build a tree out of them by correctly arranging parents and children

- explore the results of count and depth-first-search... are they correct?



NODE ANSWER

```
class Node:

    def __init__(self, value, children=None):
        self.value = value
        if children is None:
            self.children = []
        else:
            self.children = children

    def __repr__(self):
        return "Node({}, {})".format(self.value, self.children)

    # __str__ can default to repr

    def __eq__(self, other):
        return self.value == other.value
        # do we care about children?? design choice

    def __iter__(self):
        return self.children.__iter__()

    def add_child(self, child):
        self.children.append(child)

    def count(self):
        count = 1
        for child in self:
            count += child.count()
        return count

    def dfs(self, target):
        if self.value == target:
            return True
        for child in self:
            if child.dfs(target):
                return True
        return False
```