Announcements

● Ex1 due tonight @ 11PM
● Ex2 will be released tonight
● No lab next week
  ○ "Lab 2" will be released tonight
● "Ramp up" notes have been posted
Outline

● More class design
  ○ Composition
  ○ Inheritance
  ○ Exceptions
Event

- Had "day", "month", and "year" attributes
Date

- Alternative: Date as an attribute
  - Where Date is a class of its own
- Date **has** a day
  - **Has** a month
  - **Has** a year
Date

- Has a **day, month, and year**
- Can format itself (YYYY/MM/DD)
- Created by being given a day/month/year
- Can compare two Dates to know if they're the same or not
def __init__(self, day, month, year):
    self.day = day
    self.year = year
    self.month = month
def __init__(self, day, month, year):
    self.day = day
    self.year = year
    self.month = month

Give the ability to pass in a string month (e.g. "May").
MONTH_TO_NUMBER = {'January': 1,
                   'February': 2,
                   ... }

def __init__(self, day, month, year):
    self.day = day
    self.year = year
```python
# __init__

MONTH_TO_NUMBER = {'January': 1,
                   'February': 2,
                   ...}

def __init__(self, day, month, year):
    self.day = day
    self.year = year
    if type(month) == int:
        self.month = month
    else:
        self.month = self.MONTHS_TO_NUMBER[month]
```
Month

- Store as **int** or **string**?
Month

● Store as **int** or **string**?
  ○ It's up to us!
  ○ However we store it, the rest of the code should still work properly.
  ○ Just give people a way to **use** it but not directly **access** it.
Private Attributes

● Prefix attribute name with `_`
  ○ Represents that it "should not be accessed"
● E.g. self._month
● User can still access it, but if they break something, it's on them.
Modifying/Getting Private Attributes

- Write methods
  - No matter how internal representation changes, method will still work the same
Encapsulation

- Hiding attributes and providing access through other methods
- Use **getters** and **setters**
  - Getters **get** the value
  - Setters **set** the value
__init__

MONTH_TO_NUMBER = {'January': 1,
                   'February': 2,
                   ... }

def __init__(self, day, month, year):
    self.day = day
    self.year = year
    if type(month) == int:
        self._month = month
    else:
        self._month = self.MONTHS_TO_NUMBER[month]
Setter for Month

def set_month(self, month):
    if type(month) == int:
        self._month = month
    else:
        self._month = self.MONTHS_TO_NUMBER[month]
Getter for Month

def get_month(self) -> int:
    return self._month
Month as a string internally

- Similar: Just change the mapping (int -> string)
- Convert to int in the setter
- All code can still
Encapsulation

- Useful for large projects and code used by others
- Can change how you store something without breaking client code
  - Can switch to more efficient representations
Example: Ex1

- Write a "Course class" and store the enrolled and waitlisted students
- No details on how they're stored
- Only care about how to get them and use them
- See the solution after the ex1 is over.
A university has a student list that should be private. What do you name this attribute?

a) `self.student_list`
b) `self._student_list`
d) `self.__student_list`
c) `self.___student_list___`
Answer:

b) `self._student_list`
a) self.student_list Public

b) self._student_list Private

d) self.__student_list Something else

c) self.__student_list__ "Magic method"
You have a University class used by a lot of Universities. `self.student_list` is a list, but you want to change it to a dictionary. What do you do?

a) Contact everyone and tell them to switch to a getter/setter that you've made.

b) Write a getter/setter and just let them deal with the fixing their code.

c) Regret that you didn't use a getter/setter from the start.
c) Regret that you didn't use a getter/setter from the start.
Date

- Public attributes: day, year
- Private attribute: month
- Getter and setter for month
- Want: \( \text{str(Date)} \rightarrow YYYY/MM/DD \)
return "{:04}/{:02}/{:02}".format(
    self.year,
    self._month,
    self.day)
eq

1. Check if other is a Date
   a. Return False if it's not
2. Compare day/month/year
def __eq__(self, other):
    if not isinstance(other, Date):
        return False
def __eq__(self, other):
    if not isinstance(other, Date):
        return False
    return str(self) == str(other)

__str__ is called via the str() function!
str(self) == self.__str__()
Date

- __init__
- __str__
- __eq__
- Getters/setters for Month
- Public attributes Day, Year
Event

def __init__(self, day, month, year, desc):
    self.day = day
    self.month = month
    self.year = year
    self.description = desc
Event

def __init__(self, day, month, year, desc):
    self._date = Date(day, month, year)
    self.description = desc

Have to change any client code that uses self.day, self.month, self.year!
Make getters

- For the day, month, and year
  - Even if we change how Date is stored again, no client code will break.
Getting Day and Year

def get_day(self):
    return self._date.day

def get_year(self):
    return self._date.year
Getting Month

def get_month(self):
    return self._date.get_month()
def change_date(self, new_day, new_month, new_year):
    self._date = Date(new_day, new_month, new_year)
Printing an Event

- Print an Event in the format:
  Date
  Description
```python
def __str__(self):
    return "{{}}\n{{}}".format(
        str(self._date),
        self.description)
```
Composition

- Using another class as an attribute of a class
- Our class is composed of another class
- It has another class
  - Attributes are a "has a" relationship!
Examples

● A course has students
  ○ Students have student numbers, names, lives, grades, etc. (not for ex1, though)

● A computer has a CPU, RAM, etc.
  ○ These have their own manufacturers, etc.
  ○ They make up/compose a computer!
ShopCatalogue

● Gave you client code
● Given name, quantity, and price: can add items to the catalogue.
  ○ Adjust quantity by adding/removing
ShopCatalogue

- Solution is on course website.
- Represents an Item via a tuple (quantity, price)
- Gross!
  - Hard to understand what `self.items[name][0]` is!
Item

- A ShopCatalogue **has** items
- Items **have** names, prices, quantities
def __init__(self, name, price, quantity):
    Private or public attributes?
def __init__(self, name, price, quantity):
    self._name = name
    self._price = price
    self._quantity = quantity
Getters and setters

- Simply set or return the values. e.g.:

```python
def get_price(self):
    return self._price

def set_price(self, price):
    self._price = price
```
def __str__(self):
    return "{} (x{}) for {:.2f} each".format(
        self._name, self._quantity, 
        self._price)
ShopCatalogue

● Fix code to use Item instead of the tuple
  ○ Any call to self.items needs to be fixed
● Keep the dictionary (name: item) mapping
● Fix the docstring examples
def add_item(self, name, price, quantity):
    
    >>> s = ShopCatalogue("UofT")
    >>> s.add_item("Chips", 0.99, 3)
    >>> s.items
    {'Chips': (3, 0.99)}
__repr__ for Item

- Currently just the memory address
- **Want:** Name (Quantity): Price

```python
def __repr__(self):
    return "{} ({}): {:.2f}".format(
        self._name,
        self._quantity,
        self._price)
```
Fixing Docstrings

def add_item(self, name, price, quantity):
    
    >>> s = ShopCatalogue("UofT")
    >>> s.add_item("Chips", 0.99, 3)
    >>> s.items
    {'Chips': Chips (3): 0.99}
def add_item(self, name, price, quantity):
    if name in self.items:
        self.items[name][0] += quantity
    else:
        self.items[name] = [quantity, price]
Fixing Methods

def add_item(self, name, price, quantity):
    if name in self.items:
        self.items[name].set_quantity(
            self.items[name].get_quantity() +
            quantity)
    else:
        self.items[name] = [quantity, price]
def add_item(self, name, price, quantity):
    if name in self.items:
        self.items[name].set_quantity(self.items[name].get_quantity() + quantity)
    else:
        self.items[name] = Item(name, price, quantity)

Could write a method to update the quantity, if you want.
Fixing Methods

def add_item(self, name, price, quantity):
    if name in self.items:
        self.items[name].set_quantity(self.items[name].get_quantity() + quantity)
    else:
        self.items[name] = Item(name, price, quantity)

Create new Item object for new mappings.
get_items_below

below_list = []
for name in self.items:
    if (self.items[name][1] < price and
        self.items[name][0] > 0):
        below_list.append(name)

if not below_list:
    return []

below_list.sort()
return below_list

Use getters instead of indices!
get_items_below

below_list = []
for name in self.items:
    if (self.items[name].get_price() < price and
        self.items[name].get_quantity() > 0):
        below_list.append(name)

if not below_list:
    return []

below_list.sort()
return below_list
remove_item

- Use getters and setters again

```python
self.items[name].set_quantity(
    self.items[name].get_quantity() -
    quantity)
```
items_str = []
for name in self.items:
    quantity = self.items[name][0]
    if quantity > 0:
        items_str.append("{} (x{}) for {:.2f} each".format(name, quantity, price))

return "{} has: {}".format(self.name, ", ".join(items_str))
__str__

items_str = []
for name in self.items:
    quantity = self.items[name][0]
    if quantity > 0:
        items_str.append(str(self.items[name]))

return "{} has: {}".format(self.name, ", ".join(items_str))
__str__

items_str = []
for name in self.items:
    quantity = self.items[name][0]
    if quantity > 0:
        items_str.append(str(self.items[name]))

return "{} has: {}".format(self.name,
                            ", ".join(items_str))
items_str = []
for name in self.items:
    quantity = self.items[name].get_quantity()
    if quantity > 0:
        items_str.append(str(self.items[name]))

return "{} has: {}".format(self.name,
                           ", ", ").join(items_str)
ShopCatalogue

- Now uses Item class
  - Items have names, prices, quantities
- Uses *composition*
- Code is easier to read now
Inheritance

But first: a 10 minute break!
Docstring for "magic" class methods required?

I was wondering if a docstring is required for "magic" class methods like __str__? I did a google search and some say it's not necessary as python has its own description for the method, but I just wanted to clarify if this course specifically requires us to make a docstring for it.

Thanks for the segue! :)
Inheritance

- Any classes you write inherit Python's default behaviour from the object class.
- __str__ and __repr__ have default behaviour defined in object.
Inheritance

- Taking the behaviour (attributes, methods) from another class is inheritance
- We inherit behaviour from parent classes (or super classes).
  - Inherited by child or sub classes
Inheritance

● Default: Inherit \_\_str\_, \_\_repr\_, \_\_eq\_, \_\_init\_ from Object.
  ○ Everything is a subclass of object.
● Everything \textit{is} an object
● Can inherit from classes we design
Animal

- Only has an x, y, z attribute (its position)
- Animals can move
- All animals start at x = 1, y = 1, z = 1
  - For simplicity!
class Animal:
    def __init__(self):
        self.x = 1
        self.y = 1
        self.z = 1

    def move_forward(self):
        self.z += 1
Inheritance

- "Is a" relationship
  - A string is an object
  - A cat is an animal
    - Everything an animal can do, a cat can do.
  - A student is a person
    - Everything a person can do, a student can do.
Cat

- A Cat is an Animal
- In this example: just uses all the same behaviour that we defined in Animal
class Cat(Animal):

Super class
class Cat(Animal):
    pass
A Jaguar is a Cat
  ○ Inherit the behaviour from Cat
● Suppose it moves 2x faster than a Cat

```python
class Jaguar(Cat):
```

Extending methods

def move_forward(self):
    # Want to call our super class' move_forward twice,
    # extending its behaviour
Extending methods

def move_forward(self):
    # Cat.move_forward(self) is also okay!
    super().move_forward()
    super().move_forward()

We do everything our superclass does and more!
def move_forward(self):
    # Cat.move_forward(self) is also okay!
    super().move_forward()
    super().move_forward()
Extending methods

def move_backward(self):  
    super().move_backward()  
    super().move_backward()

def move_left(self):  
    super().move_left()  
    super().move_left()
Sloth

- A Sloth is an Animal
- Doesn't move

class Sloth(Animal):
Overriding methods

- We **override** the move methods
- Redefine them completely; don't refer to the superclass at all.

```python
def move_forward(self):
    return
```
Pigeon

- Can fly up or down
  - Otherwise, functionally the same as an Animal
- A Pigeon is an Animal
Extending a Class

- Subclasses can add their own methods
  - Does everything their superclass can do and more
- Simply done by implementing more methods
  - I.e. how we write classes normally
Pigeon: fly_up, fly_down

def fly_up(self):
    self.y += 1

def fly_down(self):
    self.y -= 1
Butterfly

- Can fly up and down
- Similar to a Pigeon
  - Behaves exactly the same as a Pigeon!

class Butterfly(____):

Butterfly

- A Butterfly is not a Pigeon!
- Both of them are flying animals, but one is not the other
- FlyingAnimal class!
class FlyingAnimal(Animal):
    def fly_up(self):
        self.y += 1

    def fly_down(self):
        self.y -= 1
Inheriting from FlyingAnimal

class Pigeon(FlyingAnimal):
    pass

class Butterfly(FlyingAnimal):
    pass
Inheritance

* Right now, we know how to:
  - Override
  - Extend
  - Inherit
class School:
    def __init__(self, name):
        self.name = name

class University(School):
    def __init__(self, name, motto):
        a) self.name = name
        self.motto = motto
        b) super().__init__(name)
           self.motto = motto
        c) super.__init__(name)
           self.motto = motto
        d) super().__init__(self, name)
           self.motto = motto
Answer:

b) `super().__init__(name)`
`self.motto = motto`
| a) self.name = name               | b) super().__init__(name)       |
|                                  | self.motto = motto             |
| self.motto = motto               | super().__init__(name)         |
| c) super.__init__(name)          |                               |
| self.motto = motto               | self.motto = motto             |
| d) super().__init__(self, name)  |                               |
| self.motto = motto               | super().__init__(self, name)   |

Wrong syntax!

Works, but we should use the superclass when we can!
We want animals that live in the ground: Worm, and Groundhog. Worm is defined (left). How do we implement Groundhog?

```python
class Worm(Animal):
    def move_up(self):
        self.y += 1
    def move_down(self):
        self.y -= 1
```

a) Inherit behaviour from FlyingAnimal
b) Inherit behaviour from Worm
c) Make a GroundAnimal class and inherit from that
d) Implement the move_up and move_down like we did in Worm
Answer:
c) Make a GroundAnimal class and inherit from that
Break time!
For about 10 minutes.
Animal

- We've written multiple Animal classes
  - Inherit
  - Extend
  - Override
- Let's add a new method:
  - make_sound
make_sound

- **Print** a sound (does not return a string)
- All animals should be able to make a sound: put it in our Animal class!
- How do we implement this?
make_sound

[inside class Animal]
def make_sound(self):
    print( ... )

What sound does 'Animal' make?
make_sound

[inside class Animal]
def make_sound(self):
    raise NotImplementedError
Abstract Classes

- Have unimplemented methods
- Can have functionality
- We don't want to create an instance of an Abstract Class
  - Doesn't make sense for us to create an 'Animal' or 'FlyingAnimal' -- they're too general!
Abstract Classes

- Define behaviour we want all subclasses to implement
- Similar to a template
- Any class that is an Animal should be able to call make_sound
- Override unimplemented methods!
class Cat(Animal):
    def make_sound(self):
        print("Purr")

- Class Jaguar can just inherit this!
class Sloth(Animal):
    def make_sound(self):
        print("Squeak")

class Pigeon(FlyingAnimal):
    def make_sound(self):
        print("Coo")
Exceptions

- `NotImplementedError` is an example of an Exception
- Raise errors with the `raise` keyword
- Use a `try/except` block to catch them
try/except

try:
    a = Animal()
    # We expect an error here!
    a.make_sound()
try:
    a = Animal()
    a.make_sound()
except NotImplementedError as ne:
    print(ne)
    print("Caught NotImplementedError")
Multiple except statements

```python
try:
    ...
except NotImplementedError as ne:
    print(ne)
    print("Caught NotImplementedError")
except Exception as e:
    print(e)
    print("Caught Exception")
```
Multiple except statements

```python
try:
    ...
except Exception as e:
    print(e)
    print("Caught Exception")
except NotImplementedError as ne:
    print(ne)
    print("Caught NotImplementedError")
```

Every error is an Exception, so if we have this first, no other excepts will be reached!
Other Errors

- ValueErrors
- IndexErrors
- SyntaxErrors
- You can implement your own too!
UnknownSoundException

- For Animals whose sound we don't know
  - E.g. Butterflies

```python
def make_sound(self):
    raise UnknownSoundException
```
UnknownSoundException

class UnknownSoundException(Exception):

UnknownSoundException

class UnknownSoundException(Exception):
    pass

That's all. We're done with it.

We can write an Exception that inherits from UnknownSoundException too, if we wanted to!
type() vs isinstance()

(Not "exam"/"test" material, just something you should know.)

- Last week I said "isinstance() is more generalizable than type() comparisons"
  - True, but not always the best thing to do
isinstance

- Returns True if something a subclass of something else.

```python
c = Cat()
isinstance(c, Animal) == True
```
**type(x) == type(y)**

- **More specific**
  - If you want to avoid comparisons to superclass, this is fine

- If you want to be okay with subclasses, use `isinstance()`.
Hypothetical: Grade Class

- Numeric Grade
- Letter Grade
- Numeric grades are equal iff they have the same number
- Letter grades are equal iff the have the same letter
- Letter grade vs Numeric grade?
Grade Class

- Common superclass: Grade
- Suppose we have a method in Grade: get_gpa_value
- Grade has __eq__ implemented:
  - 2 grades of different types are equal iff they represent the same GPA
  - Call Grade.__eq__ from subclass
**isinstance**

- Not symmetrical
- A Jaguar is a Cat, but a Cat is not a Jaguar
- \( c == j \) and \( j == c \) could be different!
  - Where \( c \) is a Cat and \( j \) is a Jaguar
**isinstance**

- If something fails the `isinstance` check:
  - Default to the parent's `__eq__` class
  - Or give the other `__eq__` a chance to run
    - return `NotImplemented` instead of `False`
def __eq__(self, other):
    if not isinstance(other, type(self)):
        return NotImplemented

This is not raising a NotImplementedException!
isinstance vs type(x) == type(y)

- Not something I'll put on an exam
- Just know the difference when you use it, and be careful
Homework

- No lab next week (Victoria Day)
- Might get TA office hours -- we'll see
- Will post Ex2 and "Lab" 2 later
- **Reminder: Ex1 is due at 11PM.**
  - Remember to submit your files on MarkUs!