Outline

● Introduction
● Object-Oriented Design
Your instructor

● Sophia Huynh
● Master's Student
● Undergrad was done at UofT as well (Specialist in CS)!
● First time teaching a course
  ○ Feedback is important! Let me know what I mess up!
What is CSC148?

- If 108 was about "learning how to program"
- 148 is about "how to program 'well'" (sort of)
  - Designing code/classes
  - Exploring different data types
  - Programming efficiently
Background needed for CSC148

- A credit in CSC108 is ideal
- You should know:
  - Control structures (if-statements, while-and for-loops)
  - How to define functions
  - Various types (lists, dictionaries, etc.)
- ... But maybe not in Python; that's fine!
Background needed for CSC148

- If you're nervous about some of those topics:
  - The course website has links to resources as refreshers.
  - Ask me (office hours, Piazza, after class)!
Logistics

- Course website: www.cdf.utoronto.ca/~csc148h/summer/
- Office hours: Thursdays @ 3 - 5 PM in BA3219
Piazza

- Use this for discussions
- Do not post code for anything that will be graded!
- If you need me to look at code: make a Private Post
Piazza

- I will **not** answer questions **within the first 12 hours** of it being posted!
  - Answering and asking questions is good for your learning!
- Exceptions: Urgent questions or things students wouldn't be able to answer.
- I'll (usually) answer within 3 days
Course Structure

● Weekly "labs" (not for marks)
● Weekly exercises (16% total)
  ○ Released on Thursdays, due the following Thursday at 11PM
● 2 Assignments (22% total)
● 2 Term tests (24% total)
● 1 Final exam (38%) - Need 40% to pass
Labs

- Handouts on the course website
- Monday 6 - 8PM are "lab" times
  - Attendance is not mandatory
  - Go there to ask your TA questions, work through material, etc.
- Keeps you up to speed with the course and gives you practice.
Exercises

- 1 week to do them
- 10 exercises in total
- Weighting:

<table>
<thead>
<tr>
<th>Highest Grade</th>
<th>3%</th>
<th>3%</th>
<th>2%</th>
<th>2%</th>
<th>2%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lowest Grade</td>
<td>2%</td>
<td>1%</td>
<td>1%</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>
Exercises

- Do not plagiarise!
- They're worth like 2% each. It's not worth it.
- If you need help, just ask (TAs, on Piazza, office hours, etc.)
  - The exercises are all new! Google won't help you too much.
Assignments

- 2 in total
- Highest worth 14%
- Lowest worth 8%
- A1 will be released around May 26
Assignments

● As always: **don't plagiarise!**
● There'll be plenty of office hours, labs, resources, etc. available to help you.
Midterm Tests

- 2 in total
- Highest worth 16%
- Lowest worth 8%
- T1 is on June 7th (during lecture time)
Read the syllabus!
Ask me after class or on Piazza if you have questions about it!
Software

● Using Python (like in CSC108)
● Recommended IDE: PyCharm
  ○ Lab 1 goes over installing it.
  ○ You *can* use other things, but PyCharm makes installing libraries easier.
  ○ It's also a bit more organized.
Software

- The projector in this room won't show PyCharm well, so no demo of it.
- Lab 1 will get you familiar with it (and the TAs will help you!)
Lectures

- Slides posted after class
- Rough "script" will also be posted
  - Just a general idea of what I plan to say during class; I'm bad at ad-libbing!
Lectures

● I'll randomly call out people to answer questions!
● Left-side vs. Right-side
● You'll usually have a few minutes to discuss with people around you
Are you allowed to post assignment code on Piazza?
Answer: No!
What is the weight of your lowest exercise grade?
Right-side

Answer: 0%
Lectures

- I'll keep track of the left-side vs. right-side score
  - Just for fun. Probably.
How to do well

● Do labs, exercises, and assignments
● Write code!
  ○ Practice makes perfect! Watching and reading can only help so much.
● If you need help: Come talk to us!
  ○ Labs, office hours, Piazza, etc.
● Try to start your work early!
How to do well

● Need more practice beyond the labs and exercises?

● Let me know!
  ○ I can always try to make more examples and practice questions!
Object-oriented Design
Representing a Student

We know that students have:

- Names
- Student numbers
Representing a Student

We could represent these with variables:

\[
\text{name} = "Sophia"
\]

\[
\text{student\_number} = 1234
\]
Representing a Student

We could represent these with variables:

```python
name = "Sophia"
student_number = 1234
student = (name, student_number)
```

*We could represent a student using a tuple...*
Write a Class!
What is a Class?

● Essentially like "types"
  ○ You can write your own though!
● We've seen the **str** class, **list** class, etc.
● They have **methods** which operate on them (like functions)!
Methods

"cat".upper()
Methods

"cat".upper()

upper() affects "cat", which is an instance of a string.

type is str
A Function

\texttt{upper("cat" )}
A method of str

```
str.upper("cat")
```
A method of `str`

```python
str.upper("cat")
```

- Give the type
- Pass a string in as an argument
Implementing a Student

- Name
- Student number
Implementing a Student

class Student:
Implementing a Student

```python
class Student:
    def __init__(self):
        Python's way of specifying how to initialize a class.
```
Implementing a Student

class Student:
    def __init__(self):
        self is required here, but we can add more parameters.
Implementing a Student

class Student:
    def __init__(self, n, num):
        n = Name
        num = Student Number
Implementing a Student

class Student:
    def __init__(self, n: str, num: int) -> None:

Documentation!
Add in the types for your parameters, and the return type of the method.
Aside: Type Annotations

def f(param1: type,
    param2: type) -> ReturnType:
    """ (type, type) -> ReturnType """

@param param1 type: description
@param param2 type: description
Aside: Type Annotations

```python
def f(param1: type,
     param2: type) -> ReturnType:
    """ (type, type) -> ReturnType """
    @param param1 type: description
    @param param2 type: description
```

I use this one!

I don't care which style you use, as long as you document types somehow.
Implementing a Student

class Student:
    def __init__(self, n: str, num: int) -> None:
        """ Initialize a new student with the name n and student number num. """

""" Initialize a new student with the name n and student number num. """"
Implementing a Student

def __init__(self, n: str, num: int) -> None:
    
    """ Initialize a new student with the name n and student number num. """
Implementing a Student

def __init__(self, n: str, num: int) -> None:
    """Initialize a new student with the name n and student number num. """
    self.name = n
Implementing a Student

def __init__(self, n: str, num: int) -> None:
    
    """ Initialize a new student with the name n and student number num. """
    self.name = n
    self.student_number = num
Implementing a Student

def __init__(self, n: str, num: int) -> None:
    """ Initialize a new student with the name n and student number num. """
    self.name = n
    self.student_number = num

Now we can use self.name and self.student_number anywhere within the class to get at these values!
Implementing a Student

def __init__(self, n: str, num: int) -> None:
    """ Initialize a new student with the name n and student number num. """
    self.name = n
    self.student_number = num

Descriptive parameter names are better!
def __init__(self, name: str, student_number: int) -> None:
    """ Initialize a new student with the name name and student number student_number. """
    self.name = name
    self.student_number = student_number

Implementing a Student
Making an Instance of a Student

● How do we create a Student?
Making an Instance of a Student

Need a variable to hold it.

s = 57
Making an Instance of a Student

Put in the class name.

\[ s = \text{Student} \]
Making an Instance of a Student

Pass in the parameters (aside from self)

s = Student("Sophia", 1234)
Making an Instance of a Student

Pass in the parameters (aside from self)

```python
s = Student("Sophia", 1234)

def __init__(self, name: str, student_number: int)
    -> None:
```

60
Making an Instance of a Student

\[ s = \text{Student}("Sophia", 1234) \]

This is called our **Constructor**.

It **constructs** a Student for us by calling **__init__**!
Using attributes of our Student

s = Student("Sophia", 1234)

s.name  # Gives us 'Sophia'

s.student_number  # 1234
Using attributes of our Student

def __init__(self, name: str, student_number: int):
    self.name = name
    self.student_number = student_number

    # Lets us refer to the instance itself (otherwise, it has no name attached to it)!
Creating Students

s2 = Student("David", 2345)
s3 = Student("Danny", 1024)
Creating Students

s2 = Student("David", 2345)
s3 = Student("Danny", 1024)
s2.name # 'David'
s2.student_number #2345
s3.name # 'Danny'
s3.student_number #1024
Creating Students

s2 = Student("David", 2345)
s3 = Student("Danny")
s2.name # 'David'
s2.student_number # 2345
s3.name # 'Danny'
s3.student_number # 1024
Creating Students

# Change the name of s3
s3.name = "Jen"

s3.name # will now give us 'Jen'
Class vs Tuples

`s = ("Sophia", 1234)`
`s[0]`  
`S[1]`
Class vs Tuples

`s = ("Sophia", 1234)`

`s[0]`

`S[1]`

What if we had 20 attributes?
Class vs Dictionary

```python
s = {'name': 'Sophia',
     'Student_number': 1234}
s['name']
s['student_number']
```
Class vs Dictionary

s = {'name': 'Sophia',
     'Student_number': 1234}

s['name']
s['Student_number']

It'll get ugly fast if we have a lot of attributes. It's also tedious.
Writing Methods

- Similar to writing functions
- Within a class
- Takes `self` as a parameter
Method 'enroll'

Students can **enroll** in a course.
Method 'enroll'

class Student:
    # omitted __init__
    def enroll(self, course: str) -> None:
        return None
Method 'enroll'

class Student:
    # omitted __init__
    def enroll(self, course: str)
        -> None:

Need an **attribute** to keep track of our courses!
**Go back to __init__!**
Method 'enroll'

def __init__(self, name: str, student_number: int):
    
    """ Initialize a new student with the name name and student number student_number. """
    self.name = name
    self.student_number = student_number
    self.courses = []

A student starts with an empty list of courses.
Method 'enroll'

class Student:
    # omitted __init__
    def enroll(self, course: str) -> None:
        """ Add course to this student's list of courses. """
        self.courses
Method 'enroll'

class Student:
    # omitted __init__
    def enroll(self, course: str) -> None:
        """ Add course to this student's list of courses. ""
        self.courses.append(course)
Using 'enroll'

```python
s = Student("Sophia", 1234)
s.enroll("CSC148")
```
Using 'enroll'

```python
s = Student("Sophia", 1234)
s.enroll("CSC148")
s.courses # ['CSC148']
```
Printing Students

We want to be able to do this:

```python
>>> s = Student('Sophia', 1234)
>>> s.enroll('CSC148')
>>> print(s)
Sophia (1234)
Enrolled in: CSC148
```
Printing Students

Currently, we get this:

```python
>>> s = Student('Sophia', 1234)
>>> s.enroll('CSC148')
>>> print(s)
<__main__.Student object at 0x1037686a0>
```
Printing Students

Currently, we get this:

```python
>>> s = Student('Sophia', 1234)
>>> s.enroll('CSC148')
>>> print(s)
<__main__.Student object at 0x1037686a0>
```

Python's default: Type and Memory Address!
def __str__(self) -> str:
    """ Return the string version of this Student. ""
    >>> s = Student('Sophia', 1234)
    >>> s.enroll('CSC148')
    >>> print(s)
    Sophia (1234)
    Enrolled in: CSC148
    """
```python
def __str__(self) -> str:
    """ (omitted docstring) """
    courses = ", ".join(self.courses)
```

Get the courses, joined together by a comma and space.
\_\texttt{str\_} method

def __str__(self) -> str:
    
    """ (omitted docstring) """

courses = ", ".join(self.courses)

s = "{} ({{}})\nEnrolled in: {}"
    .format(self.name,
            self.student_number,
            courses)
Aside: str format

s = "{} ({})) \nEnrolled in: {}"
.format(self.name,  
    self.student_number,  
    courses)

s = self.name + " (" +  
    str(self.student_number) + ...
Aside: str format

s = "{} ({{}})\nEnrolled in: {}"
.format(self.name,
self.student_number,
str(self.Student_number))

.format() converts things to strings for us and replaces {} with the arguments passed in.
def __str__(self) -> str:
    courses = ", “.join(self.courses)
    s = "{} ({}))\nEnrolled in: {}"
    .format(self.name,
            self.student_number,
            courses)
    return s
_str_ method

Now we get this:

```python
>>> s = Student('Sophia', 1234)
>>> s.enroll('CSC148')
>>> print(s)
Sophia (1234)
Enrolled in: CSC148
```
Comparing Students

- Suppose we want to know if two students are the same or not

- Criteria:
  - Students are the same if they have the same student number.
Comparing Students

```python
>>> s1 = Student('Jekyll', 101)
>>> s2 = Student('Hyde', 101)
```
Comparing Students

What we want:

```python
>>> s1 = Student('Jekyll', 101)
>>> s2 = Student('Hyde', 101)
>>> s1 == s2
True
```
Comparing Students

What we get currently:

```python
>>> s1 = Student('Jekyll', 101)
>>> s2 = Student('Hyde', 101)
>>> s1 == s2
False
```
Comparing Students

What we get currently:

```python
>>> s1 = Student('Jekyll', 101)
>>> s2 = Student('Hyde', 101)
>>> s1 == s2
False
```

Python's default: Comparing the **memory addresses** of s1 and s2.
def __eq__(self, other: object) -> bool:
    
    """Return True if self and other are both Students with the same student number.
    >>> s1 = Student('Jekyll', 101)
    >>> s2 = Student('Hyde', 101)
    >>> s1 == s2
    True
    """
def __eq__(self, other: object) -> bool:
    """ (omitted docstring) """
    # Compare the types
def __eq__(self, other: object) -> bool:
    if not isinstance(other, Student):
        return False
    return True
def __eq__(self, other: object) -> bool:
    if type(other) != Student:
        return False
    return True
def __eq__(self, other: object) -> bool:
    if type(other) != Student:
        return False
    return True
def __eq__(self, other: object) -> bool:
    if not isinstance(other, Student):
        return False
    return False
def __eq__(self, other: object) -> bool:
    if not isinstance(other, Student):
        return False
    if self.student_number == other.student_number:
        return True
    return False
def __eq__(self, other: object) -> bool:
    if not isinstance(other, Student):
        return False
    return self.student_number == other.student_number

This comparison already gives us True or False!
___eq__ method

def __eq__(self, other: object) -> bool:
    if not isinstance(other, Student):
        return False
    return self.student_number == other.student_number

This format is preferred!
What is the method header to override how Python converts a class into a string?

a) `def str() -> str:`
b) `def __str__() -> str:`
c) `def str(self) -> str:`
d) `def __str__(self) -> str:`
Answer:

d) def __str__(self) -> str:
What is the body of this `__init__`?

def __init__(self, uni_name: str):
    a) uni_name = self.university_name
    b) self.university_name = uni_name
    c) university_name = uni_name
    d) uni_name = university_name
Answer:

b) `self.university_name = uni_name`
More on Methods

But first... Maybe a 10 minute break?
Methods

- __init__
- enroll
- __str__
- __eq__
Methods

- \texttt{\_\_init\_}\_
- enroll
- \texttt{\_\_str\_}\_
- \texttt{\_\_eq\_}\_

These methods are in all classes.

We \texttt{\textbf{override}} Python's default behaviour when implementing them.
Magic Methods

● Surrounded by 2 underscores on both sides
● Methods that Python uses for its operators/builtins
Magic Methods

- `__init__`, `__str__`, and `__eq__` are the ones you'll see most often.
- `__repr__` comes up sometimes.
  - Might not need to implement it, but it is useful.
If we try to look at s, we get:

```python
g>>> s = Student('Sophia', 1234)
```
If we try to look at s, we get:

```python
>>> s = Student('Sophia', 1234)
>>> lst = [s]
>>> print(lst)
[<__main__.Student object at 0x1037686a0>]
```
If we try to look at s:

```python
>>> s = Student('Sophia', 1234)
>>> lst = [s]
>>> print(lst)
[<__main__.Student object at 0x1037686a0>]
```

Python represents our class with a memory address and type.
So maybe we want:

```python
>>> s = Student('Sophia', 1234)
>>> s.enroll('CSC148')

>>> s
Sophia - 1234 - ['CSC148']
```
def __repr__(self) -> str:
    """Return a representation of this Student.

>>> s = Student('Sophia', 1234)
>>> s.enroll('CSC148')
>>> s
Sophia - 1234 - ['CSC148']"""
def __repr__(self) -> str:
    """Return a representation of this Student.
    >>> s = Student('Sophia', 1234)
    >>> s.enroll('CSC148')
    >>> s
    Sophia - 1234 - ['CSC148']"
    return "{} - {} - {}".format(self.name, self.student_number, self.courses)
Interlude: self

str.upper("cat")
Interlude: self

s = Student('Sophia', 1234)

Student.enroll(s, "CSC148")
Interlude: self

```python
s = Student('Sophia', 1234)
Student.enroll(s, "CSC148")
```

There are 2 parameters!
s = Student('Sophia', 1234)

Student.enroll(s, "CSC148")

def enroll(self, course: str) -> None:
    self.courses.append(course)
s = Student('Sophia', 1234)

Student.enroll(s, "CSC148")

class Student:
    ...
    def enroll(self, course: str) -> None:
Interlude: Documentation

- Document the types of all parameters
- Include a docstring describing **what your method does**
  - Mention parameters by name
- Include 1~2 docstring examples
  - Shows how your method should be called, and what it should return/do
Interlude: Documentation

● For classes: Include a **class docstring**
  ○ Briefly describes your class
  ○ Describe the attributes of your class

● Inside of the class:
  ○ Add type annotations for its attributes
Interlude: Documentation

● For classes: Include a **class docstring**
  ○ Briefly describes your class
  ○ Describe the attributes of your class

● Inside of the class:
  ○ Add type annotations for its attributes
class Student:
    """
    A class representing a Student.
    
    name - The name of the Student
    student_number - The student number of the Student
    """

    name: str
    student_number: int
At the start of your file/module, include a **module docstring**

- Short docstring that says what's in your file
Interlude: Documentation

- Documentation is important in practice
- Tedious, but comes naturally with practice
- Appreciate well documented code!
  - What if people never documented things like `list.extend`, `str.isalpha`, etc.? :(

Designing a Class

● When do you want to write a class?
● If you have to write a class:
  ○ What attributes should you have?
  ○ What methods?
Designing a Class

- Sometimes it's obvious:
  - Not a simple structure
  - Contains some information
  - Behaves in a certain way
  - Can't represent it nicely with basic types
Designing a Class

- Usually given a description or a problem.
- Need to determine what classes are needed.
We want to keep track of an event's details. An event has a day, month, and a year (the date on which it occurs), as well as a description. An event is able to move to another date, and can tell when it conflicts with another event.
We want to keep track of an event's details.

An event has a day, month, and a year (the date on which it occurs), as well as a description.

An event is able to move to another date, and can tell when it conflicts with another event.
Identifying Attributes

- If our class *has* something, it's probably an attribute
  - E.g. A Student *has* a name, it *has* a student number, and it *has* courses.
Identifying Attributes

We want to keep track of an event's details.

An event has a day, month, and a year (the date on which it occurs), as well as a description.

An event can move to another date, and can tell when it conflicts with another event.
Identifying Attributes

We want to keep track of an event's details.

An event has a *day, month, and a year (the date on which it occurs)*, as well as a description.

A date has a day, month, and year!

We could have a Date class!

But we won't. That's next week.
Identifying Attributes

We want to keep track of an event's details.

An event has a day, month, and a year (the date on which it occurs), as well as a description.

An event is able to move to another date, and can tell when it conflicts with another event.
Identifying Methods

- Methods are usually **actions** (verbs)
- If a class can **do** something, then that something might be a method
  - E.g. A Student can **enroll** in a course.
Identifying Methods

We want to keep track of an event's details.

An event has a **day, month, and year** (the date on which it occurs), as well as a **description**.

An event is able to **move to another date**, and can **tell when it conflicts with another event**.
Method Design

change_date

● Takes in a new date to move to
● Modifies the event's date

conflicts_with

● Takes in another event to compare with
● Returns whether it conflicts or not
Implementing Event

class Event:
Implementing Event

class Event:
    def __init__(self, day, month, year, desc):
Implementing Event

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

class Event:
    ...
    def change_date(self, new_day, new_month, new_year):
        self.day = new_day
        self.month = new_month
        self.year = new_year
Implementing Event

class Event:
  ...
  def conflicts_with(self, other):
      return (self.day == other.day
               and self.month == other.month
               and self.year == other.year)
Designing from Client Code

- Given code that uses a class
- Know what methods take in and what they should return
- Internal workings are a mystery
  - "As long as it works properly, it's okay."
Example: Restaurant Menu

You are given the usage:

```python
>>> r = RestaurantMenu()
>>> r.add_item("Baguette", 3.50)
>>> r.add_item("Pizza", 5.50)
>>> r.add_item("Cola", 2.00)
>>> r.get_price("Baguette")
3.50
>>> r.get_below(4)
["Baguette", "Cola"]
```
Example: Restaurant Menu

Often this will be in the main block or
unittests like:

```python
if __name__ == '__main__':
    r = RestaurantMenu()
    r.add_item("Baguette", 3.50)
    r.add_item("Pizza", 5.50)
    r.add_item("Cola", 2.00)
    assert r.get_price("Baguette") == 3.50
    assert r.get_below(4) == ["Baguette", "Cola"]
```
Example: Restaurant Menu

Often this will be in the main block or unit tests like:

```python
if __name__ == '__main__':
    r = RestaurantMenu()
    r.add_item("Baguette", 3.50)
    r.add_item("Pizza", 5.50)
    r.add_item("Cola", 2.00)
    assert r.get_price("Baguette") == 3.50
    assert r.get_below(4) == ["Baguette", "Cola"]
```

This is called **Client Code!**
Code that uses your class.
Example: Restaurant Menu

How do we want to represent our items?

- Could use a list of tuples
  - Tuples in the form (item name, price)
- Could use a dictionary
  - Key = Item name
  - Value = price
Example: Restaurant Menu

How do we want to represent our items?

● Could use a list of tuples
  ○ Tuples in the form (item name, price)

● Could use a dictionary
  ○ Key = Item name
  ○ Value = price

Dictionaries are easier for this, so let’s use that.
Example: Restaurant Menu

class RestaurantMenu:
Example: Restaurant Menu

class RestaurantMenu:
    def __init__(self):
        self.items = {}
Example: Restaurant Menu

class RestaurantMenu:
    def __init__(self):
        self.items = {}

    def add_item(self, item, price):
        self.items[item] = price
Example: Restaurant Menu

class RestaurantMenu:
    ...
    def get_price(self, item):
        return self.items[item]
def get_below(self, price):
    items_below = []
    for item in self.items:
Example: Restaurant Menu

def get_below(self, price):
    items_below = []
    for item in self.items:
        if self.items[item] < price:
            items_below.append(item)
Example: Restaurant Menu

def get_below(self, price):
    items_below = []
    for item in self.items:
        if self.items[item] < price:
            items_below.append(item)
    items_below.sort()
Example: Restaurant Menu

def get_below(self, price):
    items_below = []
    for item in self.items:
        if self.items[item] < price:
            items_below.append(item)
    items_below.sort()
    return items_below
Things to consider

● Lots of cases:
  ○ E.g. "What happens if get_price is given an item that wasn't added"?

● Could raise an error or return -1

● If not specified, it's up to you
  ○ Maybe have a precondition to assume that won't happen
  ○ For assignments/exercises: Ask!
Homework

● Read the syllabus!

● Lab 1
  ○ Install Pycharm and get familiar with it
  ○ Practice in class design
  ○ Getting used to PythonTA

● Exercise 1 (due next Thursday @ 11PM)