Lecture 12

Queries

1 Classes

So far, we've seen several data types. But real world things are hard to store in most of these. What if I want to represent a person in code? I could keep each piece as a separate variable (e.g., name, birth date, country of origin, et cetera). But that quickly becomes unmanageable.

Let's start a little smaller and then work our way up. Let's make a 2D point type! We could represent the x & y coordinates separately as we've seen before, but we can do better; we should store them together!

```
# When defining new types, we'll typically start off with a capital
# letter.
class Point:
   """A class to model a 2D point.

   Attributes: x and y"""

   def __init__(self, x, y): # This is where the magic happens!
      self.x = x
      self.y = y
```

We've defined a new data type!!! *But*, we still haven't made any points. The class definition can be thought of as blueprint. We've described what a point will be, now we just have to make one. Let's do it!

```
import point

# We call this instatiating an instance of a class.
my_point = point.Point(1.2, 3.4) # Draw picture!!!

print("x-coord:", my_point.x)
print("y-coord:", my_point.y)
```

And similar to before, we can do a fancier import.

```
from point import Point

my_point = Point(1.2, 3.4)

print("x-coord:", my_point.x)
print("y-coord:", my_point.y)
```

2 Classes 2

Let's do another example, this time we'll model a pet.

```python
class Pet:
    """A class to model a pet.

    Attributes:
        name (str): The name of the pet.
        animal (str): The type of animal.
        age (int): The age of the pet.
    """
    def __init__(self, name, animal, age): # Later add age=0
        self.name = name
        self.animal = animal
        self.age = age
```

We can use the `Pet` class the same way that we used the `Point` class.

```python
from pet import *

my_pet = Pet("Spot", "Dog", 8)

print("I have a pet named {:s} that is a {:s} and is {:d} years old."
      .format(my_pet.name, my_pet.animal, my_pet.age))
```

3 Object Equality

Let's look at some code!

```python
from point import Point
p = Point(1, 2)
q = Point(1, 2)

print(p == q) # What do we think will happen?
```

When we define a new type, we have to tell Python what it means to be "equal". By default, Python has no idea; we have to tell it!

```python
# Add to point.py
def __eq__(self, other):
    return self.x == other.x and self.y == other.y
```

Show that this works. But what about floats! Show that this doesn't work. Sometimes we want to be a little looser with our definition of "equal". Let's write an almost equal!

```python
def epsilon_equal(v1, v2, epsilon=0.00001):
    return abs(v1 - v2) < epsilon
```

Use in `__eq__`.

4 Exceptions

Things go wrong. And we need to be able to deal with that. What happens when I do the following?

```python
print(1 + "2")
```

It crashes! More specifically, it raises an exception! More specifically, it raises a `TypeError` (this will be important in a moment). But what if I don't want the program to crash just because of a slight error? I can tell python to 'try' to do something and do something else of an exception is raised.

```
try:
    print(1 + "2")
except TypeError:
    print("Uh oh...")
```

Here I'm saying to 'try' to do a possibly unsafe operation, and if a `TypeError` occurs, instead do something else.

Let's look at a practical usage! Let's write some code that asks the user for an integer. If the user doesn't enter an integer, we'll display an error message and ask again.

```
is_valid_input = False

while not is_valid_input:
    try:
        value = int(input("Enter an integer: "))
        is_valid_input = True
    except ValueError:
        print("ERROR: Please enter an integer.\n")

print("You entered: {:d}.".format(value))
```

We'll go a little more in depth with this later (after midterm).

Finding Substrings

Go over **str**.`find`.

Project 4

Intro Project 4.