CSC148 winter 2016

Introduction to computer science

week 1

Danny Heap

heap@cs.toronto.edu

BA4270 (behind elevators)

better than phone

face-to-face best!

http://www.cdf.toronto.edu/~csc148h/winter/

416-978-5899

reading:

http://www.cdf.toronto.edu/~csc148h/winter/148Notes.pdf

other sources may be posted

January 12, 2016

Outline

Introduction

object-oriented design

What's CSC148 about?

Not about this - we think you already know this?

- well first, CSC108 was about `if` statements, loops, function definitions and calls, lists, dictionaries, searching, sorting, classes, documentation style. So you've got all that down...

  - if you didn't recently take 108?

- ...otherwise, sign up for the CSC148 ramp-up session by mailing: `<csc148w16rampup@cs.toronto.edu>`

remember

The session will be in WB116 (Wallberg 116), Saturday January 16th, 10 a.m.–4 p.m., and there is space for 240 students...

But what's CSC148 about?

- how to understand and write a solution for a real-world problem
- abstract data types (ADTs) to represent and manipulate information — hide implementation
  — expose API
- recursion: clever functions that call themselves
- exceptions: how to deal with unexpected situations
- design: how to structure a program
  we'll start with some OOP
- efficiency: how much resource (time/space) does a program use? Some programs aren't feasible

How's this course run?

clickable, on a PDF

All answers in course information sheet. Spoiler alert: meaning of life is 42. . .

after week #1, we are
bound by this

python infested by objects

how our programming language organizes 0s and 1s . . .

Here are some built-in objects to fool around with:

```
>>> w1 = "words"
>>> w2 = "swords"[1:]
>>> w1 is w2
>>> w1 == w2
>>> w1 * w2
>>> import turtle
>>> t = turtle.Turtle()
>>> t.pos()
(0.00,0.00)
>>> t.forward(100)
```

try these

objects have

value they represent

id where they are

type what sort of thing they can do

vandalizing existing classes

this is deeply wrong, except for teaching purposes…

```
>>> from turtle import Turtle
>>> t1 = Turtle()
>>> t1.pos()
(0.00,0.00)
>>> t1.forward(100)
>>> t1.pos()
(100.00,0.00)
>>> t1.neck
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Turtle' object has no attribute 'neck'
>>> Turtle.neck = "very reptilian"
>>> t1.neck
'very reptilian'
```

→ class that's standard Python module.

behaviour + data

in Python we can add things on the fly — a good thing?

Design a new class

Somewhere in the real world there is a description of points in two-dimensional space:

> *In two dimensions, a point is two numbers (coordinates) that are treated collectively as a single object. Points are often written in parentheses with a comma separating the coordinates. For example, (0, 0) represents the origin, and (x, y) represents the point x units to the right and y units up from the origin. Some of the typical operations that one associates with points might be calculating the distance of a point from the origin, or from another point, or finding a midpoint of two points, or asking if a point falls within a given rectangle or circle.*

Find the most important noun (good candidate for a class. . .), its most important attributes, and operations that sort of noun should support.

class Point
attributes x, y
methods distance_to_origin, add

build class Point…

in that deeply wrong, but informative, way

```
>>> from math import sqrt
>>> class Point:
...     pass
...
>>> def initialize(point, x, y):
...     point.x = x
...     point.y = y
...
>>> def distance(point):
...     return (point.x**2 + point.y**2) ** (1 / 2)
...
>>> Point.__init__ = initialize
>>> Point.distance = distance
>>> p2 = Point(12, 5)
>>> p2.distance()
13.0
>>>
```

module-level function, not indented under class Point

also module-level

those are methods of class —

reference to point uses "." notation

Point.distance(p2)

build class Point. . . properly!

Define a class API:

1. choose a class name and write a brief description in the class docstring.

   Point

   """a 2-D point.

2. write some examples of client code that uses your class

   Gives you ideas of how you'll use it

3. decide what services your class should provide as public methods, for each method declare an API[1] (examples, header, type contract, description)

   Do <u>not</u> write bodies of any methods yet!

4. decide which attributes you class should provide without calling a method, list them in the class docstring

[1] use the CSC108 function design recipe

continue building class Point... properly!

Implement the class:

These can save you in surprising situations

1. body of special methods __init__, __eq__, and __str__
2. body of other methods

see point.py to compare to your result

3. testing (more on this later)

rational fractions

Next week ↓

Although python has a built-in type for floating-point numbers, there is no built-in type for representing rational numbers:

> *Rational numbers are ratios of two integers p/q, where p is called the numerator and q is called the denominator. The denominator q is non-zero. Operations on rationals include addition, multiplication, and comparisons:*
> $>, \quad <, \quad \geq, \quad \leq, \quad =.$

. . . so we'll have to create our own Rational class.

build class Rational

Define a class API:

1. choose a class name and write a brief description in the class docstring.
2. write some examples of client code that uses your class
3. decide what services your class should provide as public methods, for each method declare an API (examples, header, type contract, description)
4. decide which attributes your class should provide without calling a method, list them in the class docstring

continue building class Rational

Implement the class:

1. body of special methods `__init__`, `__eq__`, and `__str__`
2. body of other methods
3. testing (more on this later)

managing attributes **num** and **denom**

Suppose that client code written by billions of developers uses Rational, but some of them complain that that class doesn't protect them from silly mistakes like supplying non-integers for the numerator or denominator, or even zero for the denominator. . .

After you have already shipped class Rational, you can write methods **get_num**, **set_num**, **get_denom**, and **set_denom**, and then use **property** to have Python use these functions whenever it sees **num** or **denom**

weird things

- what happens if, after declaring Point, you try

  ```
  print(Point.x)
  ```

  OR

  ```
  Point.y = 17
  ```

- methods can be invoked in two equivalent ways:

  ```
  p = Point(3, 4)
  p.distance_to_origin()
  5.0
  Point.distance_to_origin(p)
  ```

  in each case the first parameter, conventionally self, refers to the instance named **p**