Welcome to CSC 148!

Introduction to Computer Science

Bogdan Simion

bogdan@cs.toronto.edu

http://www.cs.toronto.edu/~bogdan

Overview

- Course logistics
- What is CSC148 about?
- Brief Python review
- Introduction to Object-Oriented Programming

Administrivia

- Instructor Contact:
  - Email: bogdan@cs.toronto.edu (please include CSC148 in the subject)
  - Office: BA 4268
  - Office Hours: Monday, 11:30AM - 1PM @TBD
- Webpage:
  - http://www.cdf.toronto.edu/~csc148h/winter
  - All course materials posted here
- Discussion board - Piazza:
  - Linked from course webpage. Read, ask questions, collaborate (do not post your code!)
- Course Info Sheet (due dates, policies, etc.):
  - Linked from course webpage, MUST read carefully!

Course overview

- Subject to change until the end of next week
- Assignments x 2: due at 10PM on the due date
  - Remark requests: submit within 7 days of results being released
- Lab/exercises x 9: except weeks 1, 11, 12
  - Start in week 2
  - Sign up for labs/tutorials on ROSI/ACORN!
- Tests x 2
- Final exam
  - You must get > 40% to pass this course!
- See weights and policies on course sheet!

Active participation

- Strong evidence that people learn better or faster by doing rather than passively listening
- Ask questions, work on exercises, participate!

Assignments

MarkUs

- Start early on the assignments!
- Make sure you **can** submit and submit periodically
- Build gradually, test your code!
- Do not wait until the very last minute to submit your assignment!

Don’t Panic!

- Help is available in many forms
  - Lectures/labs: Ask questions!
  - Office hours: My time dedicated specifically to helping you
  - Piazza: collaborative
  - Email: Longer turnaround time
  - Undergraduate TA Help Center:

    http://web.cs.toronto.edu/program/ugrad/ug_helpcentre.htm

Don’t Copy!

- Academic Integrity: Plagiarism and cheating
  - Very serious academic offences
  - Clear distinction between collaboration and cheating
    - Of course you can help your friend track down a bug
    - It is never ok to submit code that is not your own!
    - Ask questions on Piazza, but **don’t add details about your solution (especially your code!)**
  - All potential cases will be investigated fully
  - Don’t post your code in public places (Github, etc.)
  - We will run plagiarism detection software!

What should I know going into CSC148?

- From CSC108: if statements, for loops, function definitions and calls, lists, dictionaries, searching, sorting, classes, documentation style.
  - We assume you know this!

- Sign up for the ramp-up session!
  - http://doodle.com/poll/tkekcg78ght8ip8c
  - Indicate which session you wish to attend

Overview - What is CSC148 about?

- How to understand and write a solution for a real-world problem
- Abstract Data Types (ADTs) to represent and manipulate information
- Recursion: clever functions that call themselves
- Exceptions: how to handle unexpected situations
- Testing: how to write maintainable, correct code
- Design: how to structure a program (some OOP)
- Efficiency: how much resources (time/space) does a program use?

Remember ...

- Write good, well-documented code!
- Test your code!
- Practice makes perfect!
  - You must get your hands dirty and try things yourselves!

Python (brief review)

Function design recipe

- CSC108 teaches a “recipe” for writing functions (and methods)
- Adapted recipe for 148:
  - 1. Write examples of calls and the expected returned values
  - 2. Write a type contract that identifies the return value and the type of each parameter
  - 3. Write the function header
  - 4. Add a one-line summary of what the function does, above the type contract
  - 5. Write the function body
  - 6. Test your function, add more examples (tricky corner cases)

The type contract

- One style of type annotation:
  - @type parameter: type
  - @rtype: type (“return type”)
- Alternative for parameter annotation:
  - @param type parameter: description
- Allows pycharm to check that your code conforms
- Exercise: design a function length_is_multiple

Docstrings

- So, steps 1, 2, and 4 form the docstring

```
def length_is_multiple(num):
   """Return whether num evenly divides
      the length of string s.
   @param str s: a string
   @param int num: a whole number
   @rtype: bool
   >>> length_is_multiple(“two”, 2)
   False
   >>> length_is_multiple(“two”, 3)
   True
   """
   return len(s) % num == 0
```

Useful docstrings

- Docstrings - guidelines
  - Describe what a function does, be specific
  - Mention all parameters by name
  - Must not include how the function works
    - No mention of local variables, implementation details (algorithms, helper methods, etc.)
- Docstrings - purposes
  - Defines an interface => callers know how to use it
  - Helps you implement the body and meet the specs
  - Helps with debugging and code maintenance

Bad docstrings

- What if docstring is not complete?
  - The caller's expectations are not met
  - Bugs are likely
- What if docstring includes implementation details?
  - The caller might make certain assumptions
  - If implementation changes, caller code may break
- Consider the docstring as specifying the contract with the user/client

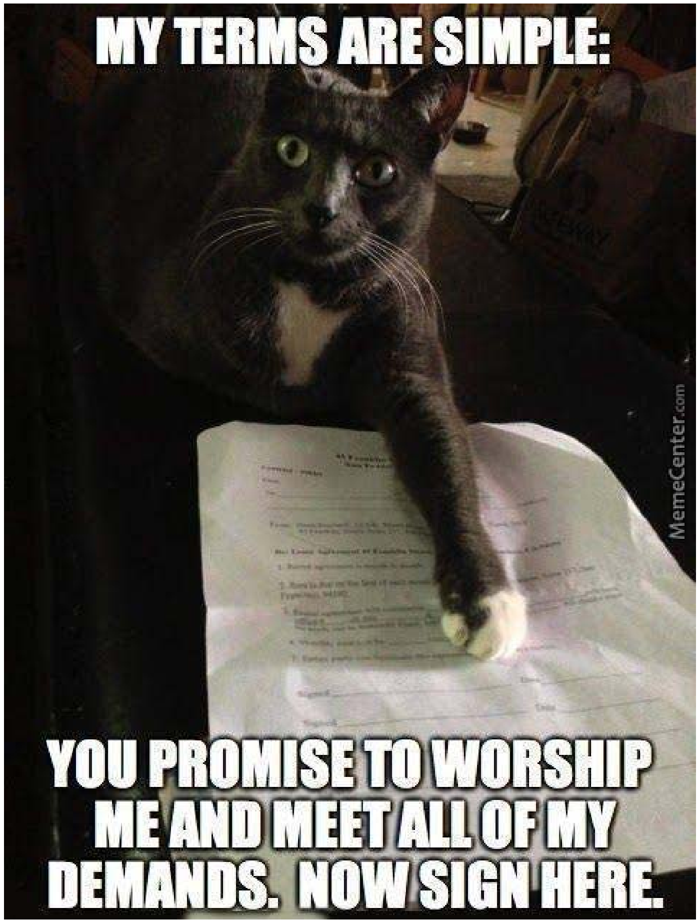
Include pre/post-conditions, if applicable

```
def square_root(number):
  """Calculate the square-root of <number>
  @type number: int
  @rtype: float

  @precondition: number >= 0
  @postcondition: abs(res * res – number) < 0.01

  <Usage examples ...>
  """
  assert number >= 0, “Uh-oh, invalid input”
  res = sqrt(number)
  assert abs(res * res - number) < 0.01
  return res
```

- Note: some design-by-contract recommend *@precondition*, *@postcondition* as decorators (more on decorators later)

Design contract - summary

- A binding agreement with the client
- Given a set of preconditions, a set of promised results will occur
  - If not => no guarantees!
- For a function, if the arguments satisfy the type contract and the preconditions, then the function:
  - a) will not crash
  - b) produces the expected result


MY TERMS ARE SIMPLE:
YOU PROMISE TO WORSHIP ME AND MEET ALL OF MY DEMANDS. NOW SIGN HERE.
MemeCenter.com

Next time ...

- Data abstraction, objects, and class design

Data abstraction, objects

University of Toronto, Department of Computer Science

An object has 3 components

- id (a reference/alias to its address in memory)
- data type (defines what they can do)
- value
- Examples: ...

Immutable data type

- Once stored in memory, it cannot change!
  - e.g., integers, booleans, strings, etc.

- Examples: ...

Mutable data type

- A type that is not immutable
  - e.g., lists, dictionaries
- Examples: ...

Verifying equality

- Equality of values in memory: ==
- Equality of addresses in memory: *is*

- Examples: ...

Object-oriented design

Classes and objects

- What’s a class?
  - Abstract data structure that models a real-world concept
  - Describes the attributes and “abilities” (methods) of that concept (called object)
  - Example: int, str, list, etc., or user-defined: Point, Rectangle, Cat, Desk, FileReader, ColourPrinter, etc.
- What’s an object?
  - Instance of a class
  - Everything in Python is an object!

Examples

- Builtin objects
  - int, string, Turtle, etc.

Using Turtle class to draw:

```
>>> from turtle import Turtle
>>> t = Turtle()
>>> t.pos()
(0.00,0.00)
>>> t.forward(100)
>>> t.pos()
(100.00,0.00)
>>> t.right(90)
>>> t.forward(100)
>>> t.pos()
(100.00,-100.00)
```

Vandalizing the Turtle class (deeply wrong!)

```
>>> t.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'
```

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.

Design roadmap – Step1

- Analyze specification:

  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.

Design roadmap – Step1

- Analyze specification:

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.

Design roadmap – Step1

- Analyze specification:

  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.

Build class Point – first attempt

- The wrong, but informative, way ...

```
>>> 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
>>>
```

Empty class (except for special methods - use dir(Point) to see)

Design roadmap – Step2

- Step 2 – 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
    - Put this code in the “main block”
  - 3. Decide what operations your class should provide as public methods, for each method declare an API (examples, type contract, header, description)
    - Refer to Function design recipe
  - 4. Decide which attributes your class should provide without calling a method, list them in the class docstring

Build class Point API

- Onto PyCharm ...

Design roadmap – Step3

- Implement the class:
  - 1. Body of special methods:

    __init__, __eq__, __str__

    __add__ (if the object should act like a numeric entity)

  - 2. Body of other methods:

    e.g., distance_to_origin, distance, etc.

  - 3. Testing (more on this later)

Implement class Point

- Onto PyCharm …

Class design: More examples

University of Toronto, Department of Computer Science

Rational fractions

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

- Similarly, we want to design and implement a class for rational numbers.

- As before, we follow the design recipe for classes

Recall from designing Point ...

- Before you start:
  - Read the specs carefully
  - Identify:
    - Frequent nouns may be good candidate for class name
    - Properties of such nouns may be good candidates for attributes
    - Operations with such entities suggest methods
    - There are some special methods that are relevant to many classes

Rational fractions

- Specification:

  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: >, <, >=, <=, =

- So we want to create our own Rational class

Build class Rational

- Step 1 – read the specs:

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: =, <>, <, >, <=, >=

Note: Python provides special methods:

__init__, __str__,

__eq__, __ne__, __lt__, __gt__, __le__, __ge__,

__add__, __mul__, etc.

Build class Rational

- Step 2 – 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
    - Put this code in the “main block”
  - 3. Decide what operations your class should provide as public methods, for each method declare an API (examples, type contract, header, description)
    - Refer to Function design recipe
  - 4. Decide which attributes your class should provide without calling a method, list them in the class docstring

Build class Rational

- Step 3 - implement the class:
  - 1. body of special methods

    Always: __init__, __eq__, and __str__,

    Others (as needed): __add__, __mul__, __lt__, etc.

  - 2. body of other methods

Special (aka magic) methods

- Python recognizes the names of special methods such as __init__, __eq__, __str__, __add__, and __mul__ and has short-cuts (aliases) for them.
- This syntactic sugar doesn't change the semantics (meaning) of these methods, but may allow more manageable code.

- For example, suppose you create a list of Rational, and then want to sort it, or check to see whether an equivalent element is in it...   [r1, r2, r3, r4, r5].sort() => needs "<" comparison

Practice, practice, practice

- Develop other methods yourselves
  - Keep in mind the docstring contract!

- Practice coding!
  - Simply understanding

  these examples is not enough!

- Did I mention practice?