EECS 390 – Lecture 1

Introduction

Essentials

- Course website – eecs390.org
  - Syllabus
  - Schedule of Topics
  - All other course materials (Google Drive, assignments, etc.)
- Canvas
- Piazza: piazza.com/umich/winter2024/eecs390
- Calendar on website
- To contact course staff: eecs390w24@umich.edu

Announcements

- All assignments available on the website
- P1 due Monday, 1/22
- HW1 due Monday, 1/29

Agenda

- EECS 390 Overview
- Logistics
- Introduction to Programming Languages and Paradigms

Questions

- Question #1: How many programming languages should an expert programmer be fluent in?
  - A) <= 2 B) 3-5 C) 6-9 D) 10+
- Question #2: How many programming languages do you think I am fluent in?
  - A) <= 2 B) 3-5 C) 6-9 D) 10+
- Question #3: What does it mean to be "fluent" in a language?

Course Objectives

- Main goals
  - Be able to quickly learn a new programming system
  - Make better use of the programming constructs and paradigms provided by a language or system
- Purpose is **not** to learn:
  - A bunch of different languages
  - The esoteric details of a particular language
- How we'll get there
  - Learn about basic features of languages, as well as common patterns and paradigms for expressing computation and data
  - Gain practice in applying techniques to large programming projects
  - Learn how to work with complex systems (languages, libraries, codebases) without needing to know all the details

Course Overview

- **Foundations:** features common to different languages
  - names and scope, control flow, memory management, syntax, grammar
- **Functional programming:** model computation in terms of inputs and outputs of functions
  - recursion, higher-order functions, anonymous (lambda) functions, continuations
- **Data abstraction:** patterns for organizing data and code associated with that data
  - message passing, object-orientation, typing, generics, modules
- **Declarative programming:** express computation in terms of relationships between code or data
  - logic programming, constraints, dependencies
- **Metaprogramming:** code that operates on code
  - macros, code generation, template metaprogramming
- **Special topics:** TBD

Course Notes

- Course notes on the website covering all the material
  - **Required** reading assignments on schedule of topics
  - We will cover a subset of the material in lecture

Exams and Grades

- Midterm Exam
  Wed. 2/21, 7pm ET
- Final Exam
  Thu. 4/25, 1:30pm-3:30pm ET
- Check to ensure you can make these times
- We will release forms for SSD soon
- **More?** See Syllabus
  - Grade thresholds, passing thresholds, etc.

Homework	15%
Projects	46%
Midterm	19%
Final	19%
Surveys	1%

Assignments

- Three homework assignments
  - Smaller programming exercises
- Five programming projects
  - Larger programming exercises to gain deeper experience in programming paradigms
- Assignments will be submitted to the autograder
- See schedule of topics for due dates
- **All deadlines are at 8pm ET**

Projects

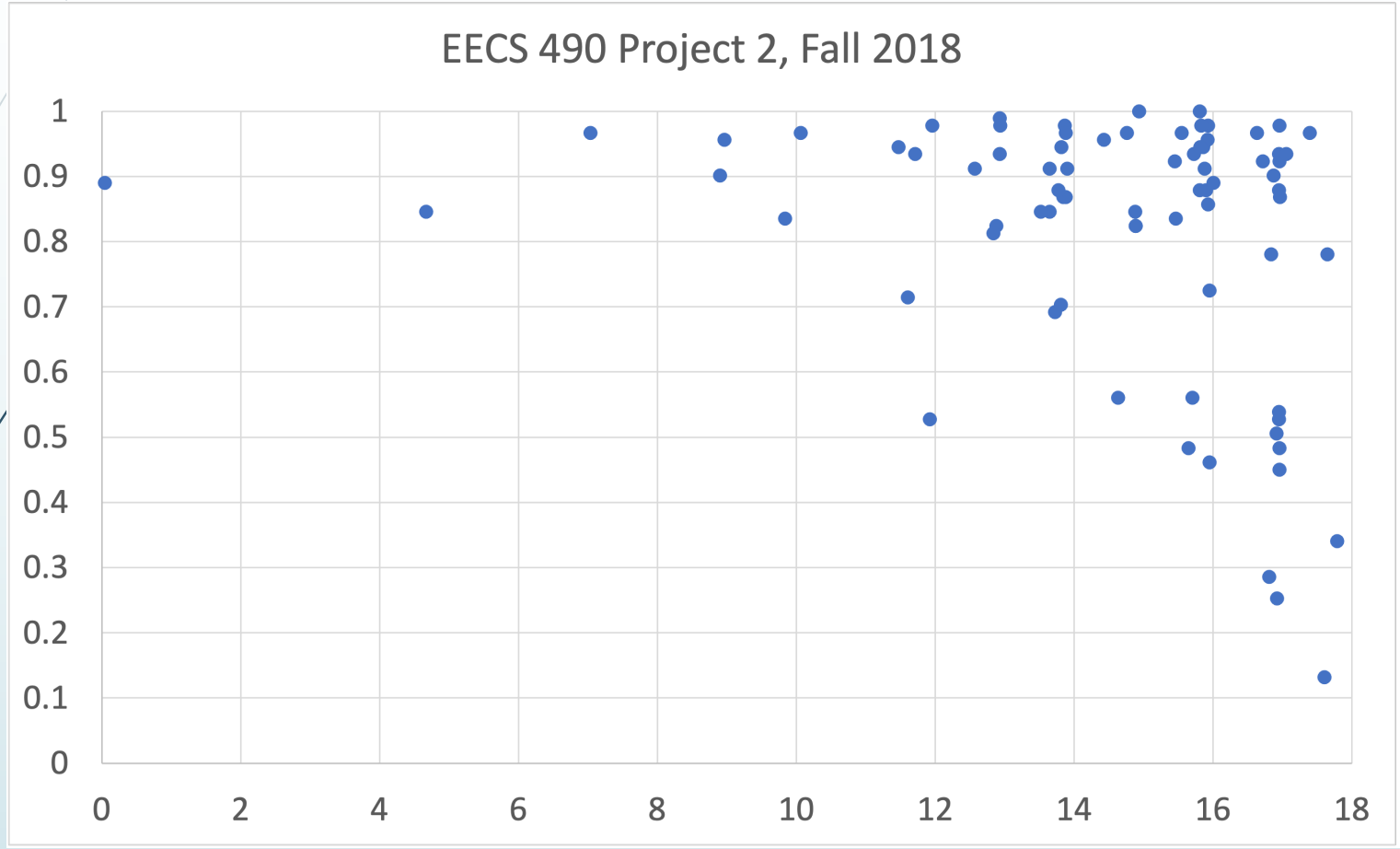
- P1: shorter project for practicing Python, reviewing abstract data types (ADTs) and object-oriented programming
- P2: Scheme parser, written in Scheme
- P3: Scheme interpreter, written in Scheme
- P4: uC static analyzer, written in Python
- P5: uC code generator, written in Python and generating C++

Project	Weight	Due
P1	6%	1/22
P2	10%	2/12
P3	10%	3/18
P4	10%	4/3
P5	10%	4/22

Project Checkpoints

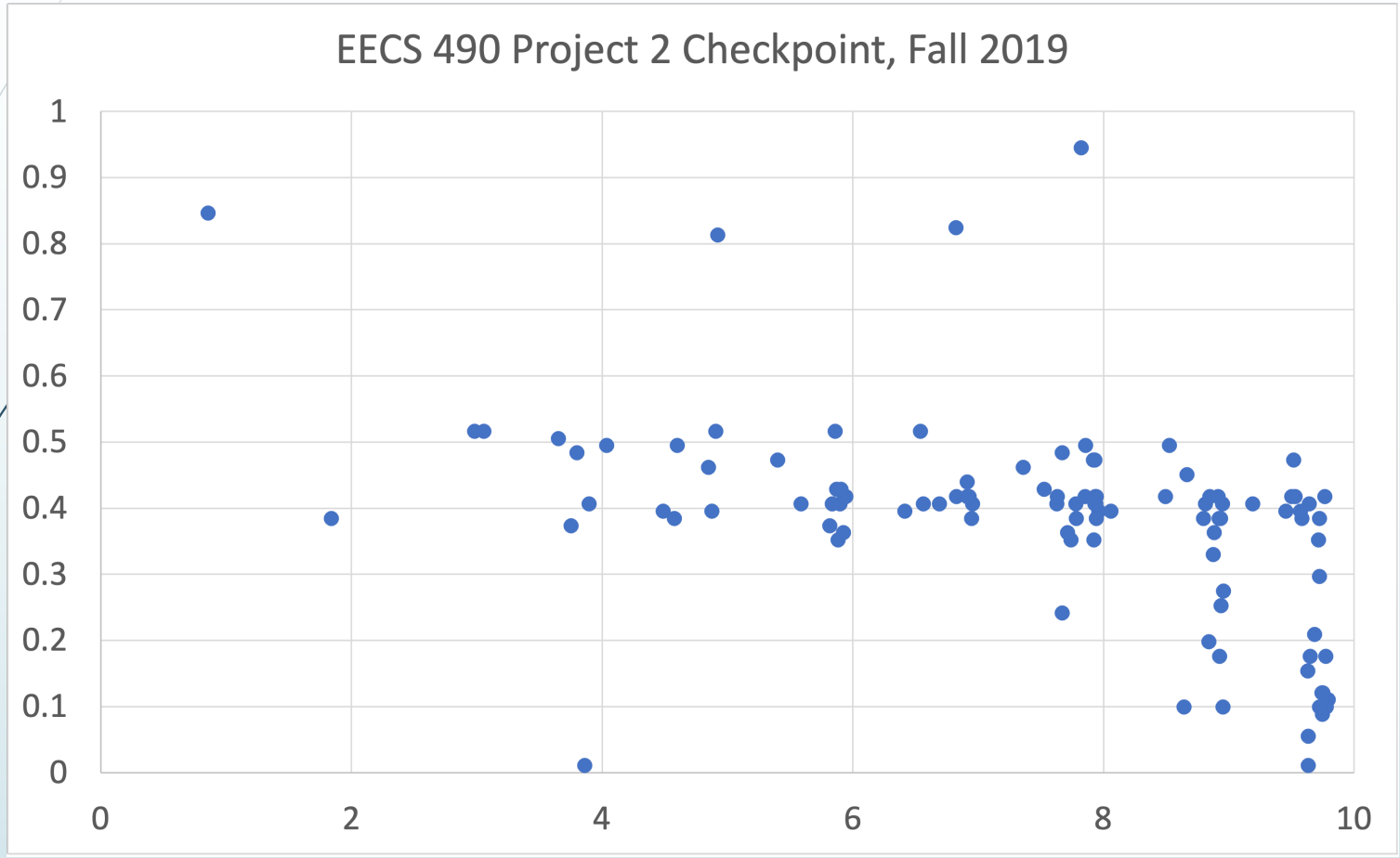
- Projects 2-5 have optional checkpoints 5-7 days before their respective deadlines
- A checkpoint is worth 20% of the points for the project
- Your score for a checkpoint is computed as follows:
  - We take your best submission before the checkpoint deadline
  - If the submission scores ≥30% of the correctness points for the project, including public and private tests, you get full credit for the checkpoint
  - Otherwise, if the checkpoint submission scores M% and your best submission before the final deadline scores N%, your checkpoint score is max(M/30,N)
- Start the projects early to ensure full credit!

First Submission Time vs. Score

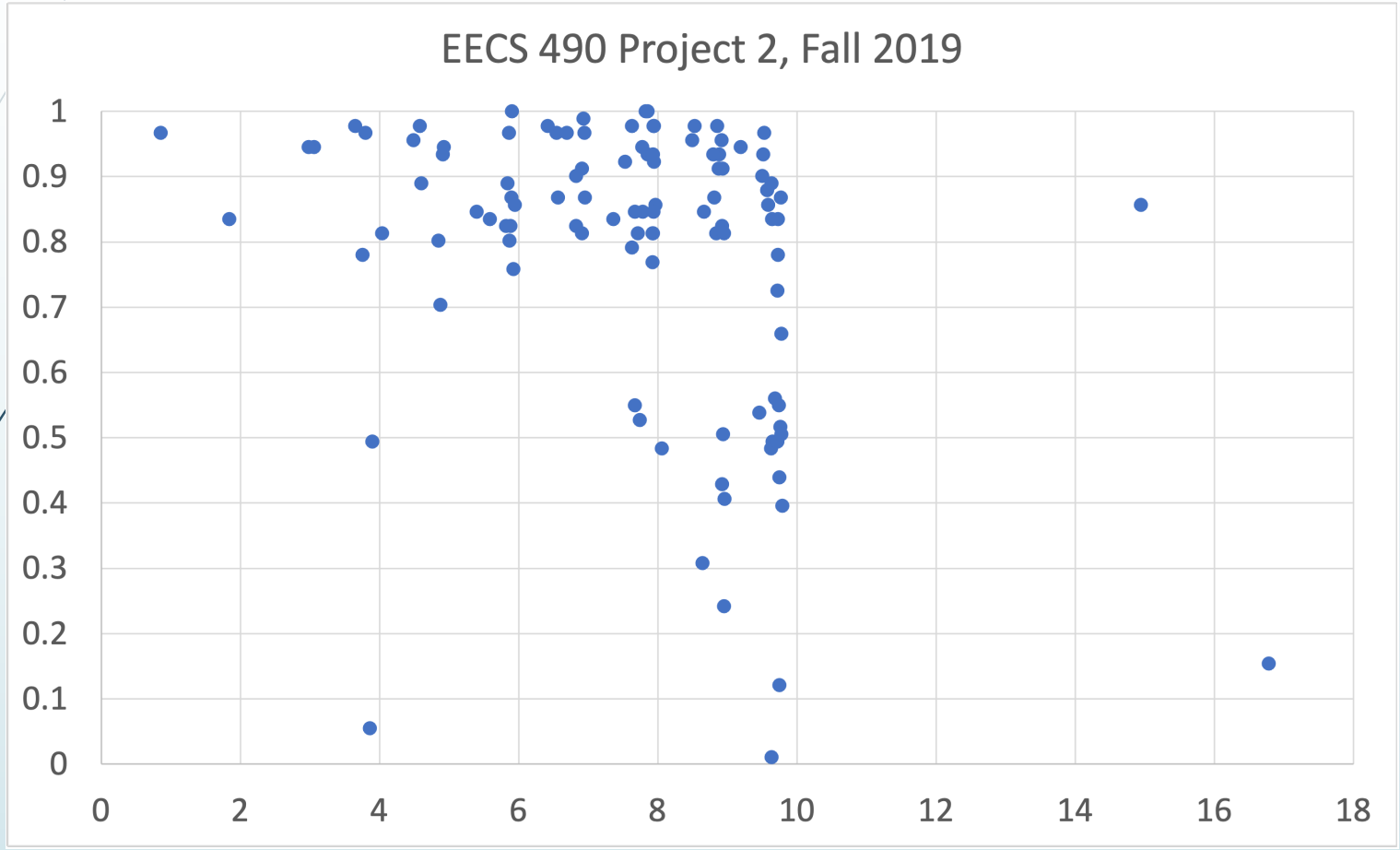


The same trend has been observed in EECS 280 and EECS 281

First Submission Time vs. Score



First Submission Time vs. Score



Collaboration

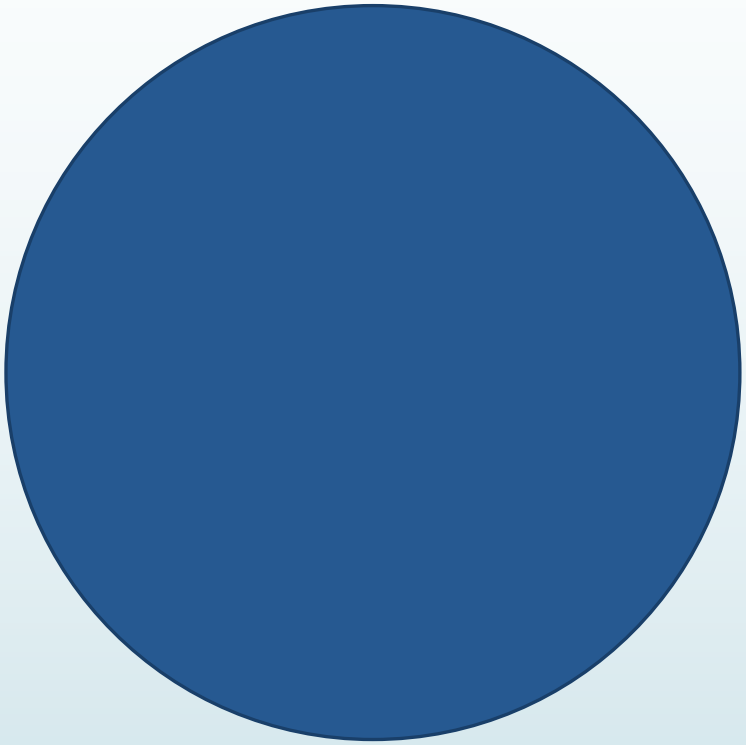
- Project 1 is to be done **individually**
- The remaining projects and all homework assignments may optionally be done with a partner
- Register your partnership for each assignment on the autograder
- Full partnership rules are in the syllabus. Please read them carefully. Following the rules is an obligation under the Engineering Honor Code
- **Partners must work on all aspects of the projects/homework together.** Both partners will be held fully responsible for any work turned in by the partnership
- No collaboration with ChatGPT, GitHub Copilot, etc.
  - See syllabus for details of what is and is not allowed

Office Hours and Piazza

- Check calendar for office hours
- Outside of office hours, post questions on Piazza
  - Please post all relevant details, e.g. what command you ran, the full content of error messages, etc.
- We will do our best to help on all aspects of the course
- Here's what we need from you:
  - Be up to date with the course material (e.g. lectures, discussions)
  - Be up to date with your own code
    - We need to understand your thought process to be able to help effectively
    - Reminder: course policy requires partners to work on all code together

Break Time!

- We'll start again in five minutes.



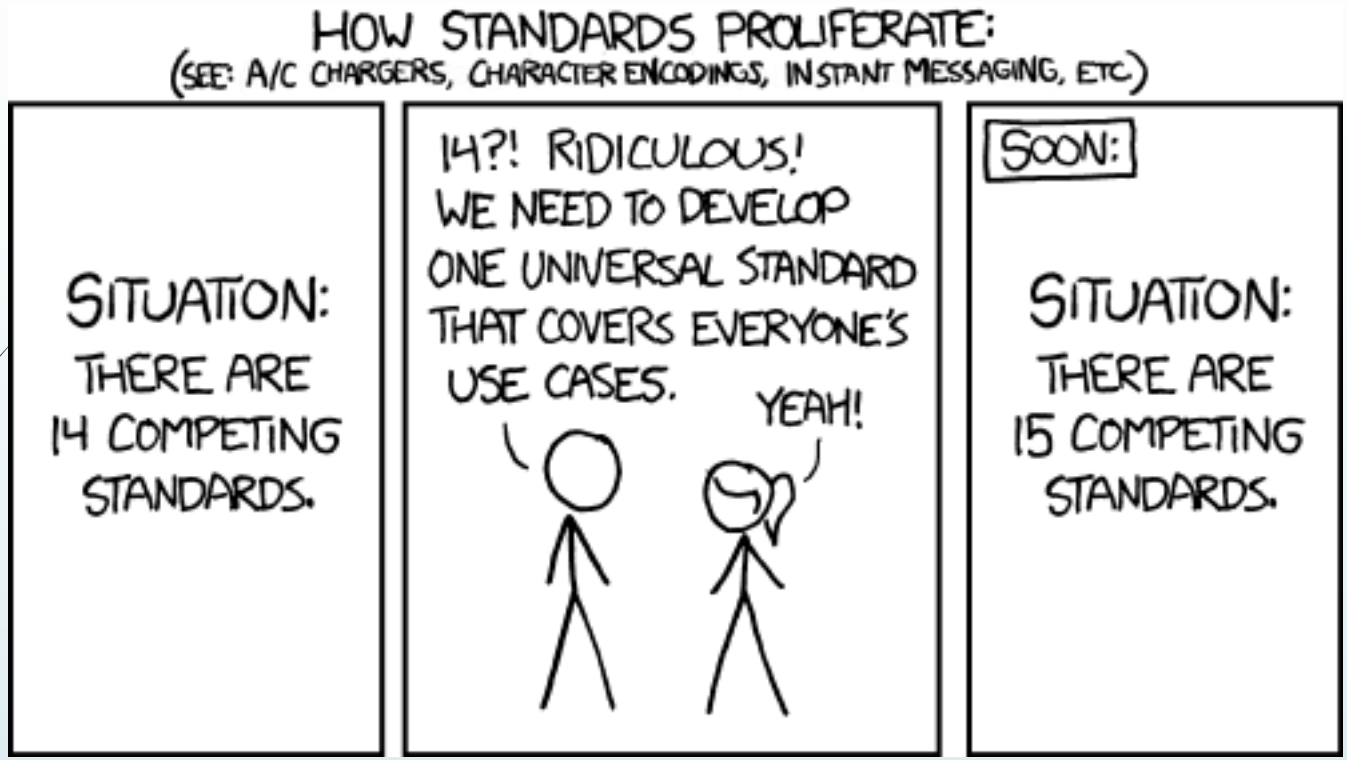
Programming Languages

- Designed for expressing computation at a higher level than machine code
  - Provide a view of computation that is ***abstracted*** from that of the machine
- Facilitate writing, reading, and maintaining code
- Provide abstractions for common programming patterns
- A common base for modules written by different programmers

Turing Completeness

- A language is ***Turing complete*** if it can compute the same functions as a Turing machine
  - Church-Turing thesis: all functions that can be computed by humans can be computed by a Turing machine
- All general-purpose languages are Turing complete
- However, languages differ in the abstractions they provide, their performance, etc.

One Language to Rule Them All?



https://xkcd.com/927/

Language Design Goals

- Some language design goals
  - Ease of writing
  - Ease of reading
  - Maintainability
  - Reliability and safety
  - Performance
  - Modularity
  - Portability

- These goals are often in conflict with each other
  - "There are no solutions; there are only trade-offs."
    – Thomas Sowell

Problem Domains

- Languages are often well-suited to a particular problem domain
  - Shell scripting: Bash
  - High-performance numerical codes: Fortran
  - Writing documents: Latex
  - Build automation and dependency tracking: Make
  - Web programming: Javascript
  - Systems programming: C
  - Etc.
- A programmer should use the right tool for the job

Programming Paradigms

- Languages can be classified in many ways
- A fundamental classification is by what programming paradigms they support
  - Imperative programming
  - Declarative programming
    - Functional programming
    - Logic programming
  - Object-oriented programming

Digression: Value and Reference Semantics

- ***Value semantics***: a variable is nothing more than a name associated with an object
  - The storage for the variable is the same as that of the object itself
  - The association between a variable and an object cannot be broken as long as the variable is in scope
- ***Reference semantics***: a variable is an indirect reference to an object
  - A variable has its own storage that is distinct from that of the object it refers to
  - A variable can be modified to be associated with a different object

Digression: Reference Semantics

- In a language with reference semantics, variables behave like C/C++ pointers
  - But can't do arithmetic on them
- Example:

```
>>> x = []
>>> y = x
>>> print(id(x))
4546751752
>>> print(id(y))
4546751752
```

Get unique ID of object

C++ equivalent:

```
list *x = new list();
list *y = x;
cout << x << endl;

cout << y << endl;
```

- Python: everything has reference semantics
- Java: primitives have value semantics, everything else has reference semantics