Learning Other Programming Languages

6.100 LECTURE 26

SPRING 2026

Announcements

- Pset 7 checkoff due next Tuesday 5/12 during office hours
- Exam 3 scores will be finalized and copied from Gradescope soon
- Review Scores page and let us know of any discrepancies by Wednesday 5/13 midnight
- Please submit subject evaluation
  - https://registrar.mit.edu/classes-grades-evaluations/subject-evaluation
  - https://eduapps.mit.edu/subjeval/studenthome.htm
  - open until Friday at 9 am (when finals begin)

Building a Language

Built-in basics

- **Built-in data types**
  - primitives: booleans, numbers, characters
  - collections: strings, arrays, hash tables, mappings
- **Turing-completeness**
  - conditionals: **`if`**
  - iteration/looping: **`while`, `for`**

Abstractions

- **Functions**
  - **behavior abstraction,** input and output for “subprograms”
  - definition and calling syntax, calling semantics
  - semantics of side-effects, mutation
- **Exceptions**
  - semantics of errors and handling mechanism
  - usually augments function execution semantics
- **Classes**
  - **data abstraction,** group data with their associated behavior
  - rich landscape of approaches

Standard libraries

- **more collections**
  - more specialized types (e.g., queues)
  - associated algorithms (e.g., sorting)
- **math** and **random** features
- interface to operating system
  - **filesystem**, networking, etc.
- **graphical display** (i.e., windows, not just text terminal)
- **concurrent execution**
  - e.g., for matrix multiplication, fill in separate quadrants simultaneously

A Brief
(and Incomplete and Opinionated)
History of Programming Languages

1940s and 1950s

- **Machine languages**
  - manually write processor instructions
  - "program" them into computer via switches
  - expensive mainframe computers, each with its own language
- **Assembly languages**
  - "syntactic sugar" for machine languages
  - "human-readable" representation of processor instructions
  - still specific to each type of expensive mainframe
- Both are still key components of modern computer architectures, but are considered "low-level" programming

Late 1950s through 1970s

- "high-level" languages
  - express technical intent
  - translate/compile down to assembly/machine language
- **Fortran** (IBM)
  - target numerical/scientific/performance computing
    - MATLAB was originally written to access Fortran libraries
  - add functions/subroutines
- **Algol** (design by committee)
  - for clean specification of algorithms
  - add block structure, recursion/stack
  - ancestor of modern syntax for variables, function calls, loops
- **Lisp** (John McCarthy, MIT)
  - "mathematical" view of programming, designed to support functions as objects, emphasize recursion
  - automatic garbage collection

1980s, 1990s

- Maturation of object-oriented programming
- **Simula:** model entities in a simulation
- **Smalltalk:** model everything as an object and communicate via “methods”
- Standardization of Lisp: **Common Lisp** vs **Scheme**
- **C++, Java:** provide large OOP libraries, adopted by industry
- **Python**
  - simplify C syntax, inspired by Lisp (among others)
  - built-in OOP semantics
  - dynamic typing (i.e., variable can point to any type of object)

2000s+

- More focus on web applications, memory safety, concurrency, performance
- **Web:** PHP, Ruby, JavaScript/TypeScript
- **Safety, concurrency:** Go, Rust
- **Scientific:** Julia
- **Platform-specific:** Kotlin (Android), Swift (iOS)

Using Different Languages

Running example

- Rename photo file names from a camera/phone
- Original naming
  - **PXL_YYYYMMDD_HHMMSSmmm.jpg**
- Desired naming
  - **YYYY-MM-DD-HHMMSS.jpg**

Python

- Top-level function
  - read file names within source directory
  - for each name, generate the renamed format
  - copy old file into target directory with renamed format
- Reformatting function
  - split into component substrings
  - recompose relevant parts into new string

C (similar syntax, different model)

- C requires a specific **`main()`** function
- All **variables and functions have types** associated with them
  - by default, variables refer to **data stored directly on stack,** not in heap
  - need to know in advance **size of data** and inputs/outputs
- Fewer standard library conveniences
  - have to implement our own copy_file() function
  - have to **close files and folders manually**
- **Strings are mutable arrays of characters**
  - no string slicing
  - more efficient to write to already allocated arrays than to pass them up and down the stack

Scheme/Racket (different syntax, similar model)

- Parentheses syntax for **"symbolic expressions"**
  - everything is such an expression, naturally expresses function call
    - `(operation argument argument ...)`
  - every expression evaluates to an object/value
- **Special forms** have different semantics depending on first elements
  - define a variable or function
    - `(define (op param ...) ...)`
  - define names within a limited scope
    - ```
      (let ((var expr)
            (var expr))
        ...)
      ```
    - evaluates to last form in the `let` body
  - iteration, same syntax as `let` but with `for` instead of `let`, and `expr` must be iterable
- Otherwise, almost identical **stack/heap model** as Python
      ```

Useful Tooling

Version control

- **Track the history** of a set of a files
  - save the state of a filesystem folder at any time
  - if messed up code, can revert back to previous state
  - can edit different versions on separate branches and merge them later
- Common tool: **git**
  - model history as a directed graph (!)
  - the backing model for online code repositories like GitHub
- **Basic commands**
  - **`init`, `clone`** – make or copy a repo
  - **`commit`, `log`, `checkout`** – save state, view history, revert to a state
  - **`branch`, `merge`** – work separately and merge into main branch
- **References**
  - https://git-scm.com/
  - https://jwiegley.github.io/git-from-the-bottom-up/

Other tools

- Install Python libraries
  - **`$ python -m pip install ...`**
  - https://pypi.org/project/pip/
- Use **virtual environments** to minimize dependency conflicts
  - https://docs.python.org/3/library/venv.html
- Use **virtual machines** to try code/programs in an isolated environment
  - **full-featured VMs:** VMWare, Parallels, VirtualBox
  - **lightweight VMs:** Docker, Windows Subsystem for Linux (WSL)
- Student-produced IAP course
  - https://missing.csail.mit.edu/

6.100 takeaways

- **Foundational programming skills**
  - gain confidence in interpreting syntax
  - gain confidence in breaking down problems and composing solutions
- Exposure to **computer science concepts**
  - stochastic simulation
  - graph modeling and search
  - recursion and dynamic programming
- **Keep learning!**
  - don’t be afraid to write small programs and fail
  - explore AI tools, be responsible, maintain technical curiosity
  - have fun programming!

Thank You and Enjoy the Summer!