Branching and Looping

6.1000 LECTURE 2

Course instructors

- **6.1000**
  - Andrew Wang
  - Tim Kraska
- **6.100A**
  - Ana Bell
- **6.100B**
  - John Guttag
- **Contacts for 6.1000**
  - 6.1000-staff@mit.edu
  - 6.1000-instructors@mit.edu

Last time

- computation, representation, objects
- types, operations
- variables
- syntax of `str` operations

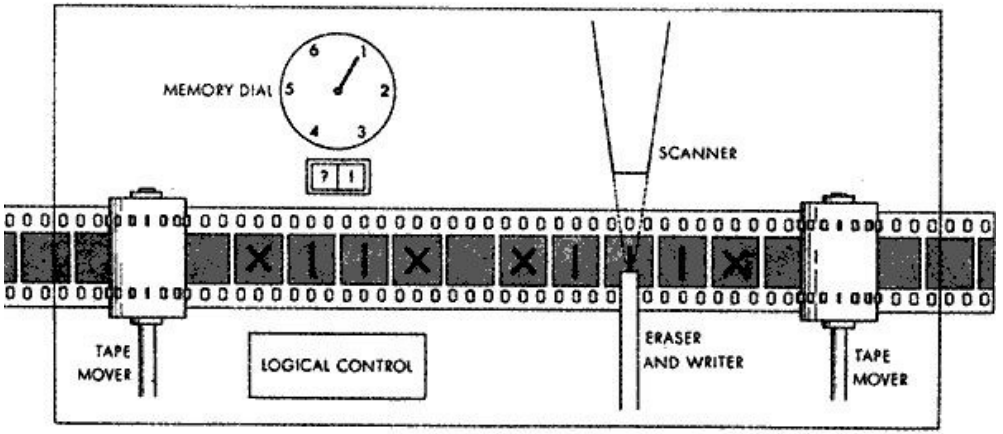
Why Python in 6.100

- Advantages
  - widely used
  - straightforward syntax
  - straightforward **semantics**
  - **focus on programming concepts,** away from hardware
  - well-designed conveniences
- Disadvantages
  - widely used
  - many features
  - well-designed conveniences

Pset 1
- released today
- due next Wed 9/17
- **use only Python features discussed in Lectures 1–3**

Computability and the Turing machine

- What mechanisms are needed to compute?
- What tasks are actually computable?
- Consider idealized hardware: **Turing machine**
  - infinite tape
  - read/write head
  - internal state
- Program is a **table lookup**
  - ends when land on a HALT state



- Each time step:
  - **Input**
    - read tape symbol
      look up current state
  - **Output**
    - write tape symbol
    - move head left/right
    - set next state

Conditionals

- Revisit square roots algorithm
  - **guess** an answer
  - compare **guess²** to input
  - **if close enough, stop**
  - **otherwise**, update new **guess = (guess + input / guess) / 2**
  - compare **guess²** to input
  - …
- Python syntax
  - ```
    if boolean expression:
        do something
    else:
        do another thing
    ```
    ```

Conditionals syntax

- No curly braces
- **Indentation matters!**
  - convention is 4 spaces
  - require **`pass`** statement in empty block
- Immediately nested **`else: if:`** can be collapsed to **`elif:`**
  - reduces indentation
- Immediately nested **`if:`** *`a`* **`if:`** *`b`* not always equivalent to **`if`** *`a`* **`and`** *`b`***`:`**

Limitations so far

- Square root algorithm
  - don't know **in advance** **how many times** to check closeness
- Deeper issue
  - **operations only:** each line gets run in sequence
  - **plus conditionals:** each line gets run **at most once**
  - **implication:** programs must be as long as all the possibilities they could compute
  - need to compactly express computations that could produce rich set of outputs

Looping with `while`

- General mechanism in Python
  - **`while`** *`condition:`*

    `body code of loop`

    `...`
- Loop exits only once *`condition`* is `False`
  - *`condition`* is an unchanging expression in code
  - but its evaluation depends on what variables it references
  - so **body code** needs to **update relevant variables**

Looping with `for`

- Python syntax
  - ```
    for var in iterable:
        Loop body code
        ...
    ```
- A **Python iterable** is a certain type of object
  - produces one value at a time specifically when "queried" by the **`for`** mechanism
  - so far, we've encountered **`str`** and **`range`** types
  - https://docs.python.org/3/library/stdtypes.html#range
- Loop automatically exits when *`iterable`* is exhausted
  - **`for`** makes repeated assignments to variable *`var`* until then
    ```

Generate-and-test

- A broad computational theme, naturally expressed with loops
- **Generate**
  - enumerate possible solutions respecting some constraints
- **Test**
  - check each candidate against remaining constraints
- Other names
  - **guess-and-check**
  - **exhaustive enumeration**
  - **brute force**

Generate-and-test scenario

- **Alyssa, Ben,** and **Cindy** are selling tickets to a fundraiser.
  - Ben sells 20 fewer than Alyssa
  - Cindy sells twice as many as Alyssa
  - 1000 total tickets were sold by the three people
- **How many tickets did each sell?**
  - could solve this algebraically
  - let's try exhaustive enumeration and testing each candidate solution

Interrupting loop execution

- `break`
  - immediately jumps out of loop
  - e.g., looking for any solution, found one
- `continue`
  - stops current loop iteration
  - hands control back to **`while`** or **`for`** to start next iteration
  - e.g., current candidate violates a constraint, no need to check remaining constraints

Generate-and-test: Bisection search

- **Scenario:** back to square roots
- Suppose didn't know original algorithm
  - could step through candidate numbers starting from 1, 1.001, 1.002, 1.003, …
  - wasteful and slow
- **Insight: n^2** increases **monotonically** with **n**
  - if **$guess^2 < query$,** then can infer **$guess < true\ root$**
  - if we chose guess wisely, can remove large chunks of candidate space

Generate-and-test: Bisection search

- **Algorithm:** maintain **lower** and **upper bounds**
  - choose guess in the middle
  - evaluate against query
  - **prune half of feasible range** by adjusting lower or upper
- **Monotonicity** is important
  - formulated for **one-dimensional** situations
- Can adapt to **discrete sequences** as well
  - find a middle index position
  - determine whether answer lies to the left, right, or on it
- Terminology
  - **bisection** (continuous) vs **binary** (discrete) search

So far

- Computation is about expressing **mechanism** to get **from input to output**
- Have provided all means necessary to express any computation
  - objects, operations, conditionals, looping
- Limitations
  - can only use variables or terminal for input/output
  - requires manual effort
  - **hard for programs to talk to each other and get reused**
- Next time
  - organize behaviors into **"modular"** subprograms
  - Python's **function mechanism**