CS 105: Introduction to Computer Science

Prof. Thao Nguyen

Spring 2025

Materials adapted from Dave Wonnacott

Recap

- *Dictionary Notional Machine*
  - Program execution is "at" (or "in") a given line
  - Moving to the next line updates "*dictionaries*" of variables (one per "*stack frame*")
  - Like PyCharm debugger or pythontutor stepping operation
- *Substitution Notional Machine*
  - Copying or replacing elements of the program until you just have one answer
  - Like some PyCharm "*refactor*" options
    - refactoring is an important activity in programming; we treat it as a notional machine
- Controlling *Abstraction level* is important in both
  - E.g., "step over" vs. "step into"

Programming Paradigms

Substitution works most naturally for *pure-functional* code, i.e., made with

- Function definitions with returned values (rather than printed results)
- Operations such as +, *, -, <=, ==, etc. (for numbers, strings, whatever)
- Variable definitions providing a value when a variable is created
- Uses of "if" that *select* a value, e.g., to put in a variable or return

We'll stick to the Dictionary Notional Machine for *imperative* code, i.e., using

- Uses of "print" and "input"
- Changing the value of a variable, including inside an "if"
- Loops (special notation for certain patterns of recursion)

Imperative Programming and Destructive Assignment

First example: "destructive assignment" is hard to understand with the S.N.M.:

```
x: int = 1    # attach the name x to the value 1
y: int = x+1  # compute x+1, attach the name y to it
x = 100       # re-attach x, to 100 instead of 1
print(x)      # can't substitute '1' for x here
print(y)      # can't substitute x+1 for y here
```

(*Note: we can fix this by re-labelling x's as x_1 and x_2 if we need to.*)

Imperative Programming and Elseless If

"destructive assignment" or "early return" are easier to understand as steps

```
def abs(x: float)->float:
    if x<0:
        x = 0-x
    return x
```

```
def abs_v2(x: float)->float:
    if x>0:
        return x
    y: int = 0-x
    return y
```

(*Once again, we can use substitution if we re-label x's, but it's harder*)

Imperative Programming and I/O (input and output)

What happens if we substitute the definition of *first_name* where it's used?

```
def get_name(greeting: str)->str:
    return input(greeting + ' ')  # or print, then input()

def example()->None:
    first_name: str = get_name("Enter your first name")
    last_name: str  = get_name("Enter your last name")
    print("your full name is", first_name + last_name)
    print("Nice to meet you", first_name)
```

Imperative Programming and While Loops

Two ways to keep trying until we get a result:

```
def get_name(greeting: str)->str:    # recursive approach
    name: str = input(greeting + ' ')
    return name if name != '' else get_name(greeting)

def get_name_loop(greeting: str)->str:  # use a while loop
    name: str = input(greeting + ' ')
    while name == '':    # empty string, try again!
        name = input(greeting + ' ')  # update name
    return name  # returns the final result
```

Imperative Programming and For Loops

Two ways to look through every element of a *collection* (e.g., a string):

```
def print_letters(name: str)->None:    # recursive approach
    if name != '':
        print(name[0], end=' ')
        print_letters(name[1:])

def print_letters_loop(name: str)->None:  # use a for loop
    for letter in name:
        print(letter, end=' ')
```

Example/exercise

As a group, or in small groups, let's write earliest_letter using a loop

- this time, *no recursion*
- which loop should we use?
- how do we update variables each time we consider a new letter?

Which approach is better/easier? It depends on the problem!
So, you need to be able to use both.

If time permits, more examples