Lecture 8

Algorithm Design
# Announcements For This Lecture

## Assignment 1
- Due **TOMORROW**
  - Due *before* midnight
  - Submit something…
  - Last revision Sep. 26
- Grades posted Friday
- Complete the Survey
  - Must answer individually

## Getting Help
- Can work on it in lab
  - But still have a new lab
  - Make sure you do both
- Consulting Hours
  - But expect it to be busy
  - First-come, first-served
- One-on-Ones still going
  - Lots of spaces available
• **Algorithm**: A step-by-step procedure for how to do something (usually a calculation).

• **Implementation**: How to write an algorithm in a specific programming language

• Good programmers know how to separate the two
  - Work out algorithm on paper or in head
  - Once done, implement it in the language
  - Limits errors to *syntax errors* (easy to find), not *conceptual errors* (much, much harder to find)

• Key to designing algorithms: *stepwise refinement*
Algorithms: Heart of Computer Science

- **Algorithm**: A step-by-step procedure for how to do something (usually a calculation).
- **Implementation**: How to write an algorithm in a specific programming language.

- Good programmers know how to separate the two:
  - Work out algorithm on paper or in head
  - Once done, implement it in the language
  - Limits errors to **syntax errors** (easy to find), not **conceptual errors** (much, much harder to find)
- Key to designing algorithms: **stepwise refinement**

Python does what you say, not what you meant

Python cannot “understand” you
Stepwise Refinement: Basic Principles

• **Write Specifications First**
  Write a function specification before writing its body

• **Take Small Steps**
  Do a little at a time; make use of placeholders

• **Run as Often as You Can**
  This can catch syntax errors

• **Separate Concerns**
  Focus on one step at a time

• **Intersperse Programming and Testing**
  When you finish a step, test it immediately
Using Placeholders in Design

• Delay do anything not immediately relevant
  - Use comments to write steps in English
  - Add “stubs” to allow you to run program often
  - Slowly replace stubs/comments with real code
• Only create new local variables if you have to
• Sometimes results in creation of more functions
  - Replace the step with a function call
  - But leave the function definition empty for now
  - This is called top-down design
Function Stubs

Procedure Stubs

- Single statement: `pass`
  - Body cannot be empty
  - This command does nothing
- **Example:**
  ```python
def foo():
    pass
  ```

Fruitful Stubs

- Single return statement
  - Type should match spec.
  - Return a “default value”
- **Example:**
  ```python
def first_four_letters(s):
    return '' # empty string
  ```

Purpose of Stubs

Create a program that may not be correct, but does not crash.
Example: Reordering a String

- `last_name_first('Walker White')` is 'White, Walker'

```python
def last_name_first(s):
    """Returns: copy of s in form <last-name>, <first-name>
    Precondition: s is in the form <first-name> <last-name> with one blank between the two names""
    # Find the first name
    # Find the last name
    # Put them together with a comma
    return ' ' # Currently a stub
```
Example: Reordering a String

- `last_name_first('Walker White')` is 'White, Walker'

```python
def last_name_first(s):
    """Returns: copy of s in form <last-name>, <first-name>
    Precondition: s is in the form <first-name> <last-name>
    with one blank between the two names""
    end_first = s.find(' ')
    first_name = s[:end_first]
    # Find the last name
    # Put them together with a comma
    return first_name # Still a stub
```
Refinement: Creating Helper Functions

```python
def last_name_first(s):
    """Returns: copy of s in the form <last-name>, <first-name>
Precondition: s is in the form <first-name> <last-name> with one blank between names""
    first = first_name(s)
    # Find the last name
    # Put together with comma
    return first # Stub
```

```python
def first_name(s):
    """Returns: first name in s
Precondition: s is in the form <first-name> <last-name> with one blank between names""
    end = s.find(' ')
    return s[:end]
```
def last_name_first(s):
    """Returns: copy of s in the form <last-name>, <first-name>
Precondition: s is in the form <first-name> <last-name> with
with one blank between names""
    first = first_name(s)
    # Find the last name
    # Put together with comma
    return first # Stub

def first_name(s):
    """Returns: first name in s
Precondition: s is in the form <first-name> <last-name> with
one blank between names""
    end = s.find(' ')
    return s[:end]

Do This Sparingly
• If you might use this step in another function later
• If implementation is rather long and complicated
Example: Reordering a String

• last_name_first('Walker White') is 'White, Walker'

```python
def last_name_first(s):
    """Returns: copy of s in form <last-name>, <first-name>
    Precondition: s is in the form <first-name> <last-name> with one or more blanks between the two names"
    # Find the first name
    # Find the last name
    # Put them together with a comma
    return ' ' # Currently a stub
```
Exercise: Anglicizing an Integer

- anglicize(1) is “one”
- anglicize(15) is “fifteen”
- anglicize(123) is “one hundred twenty three”
- anglicize(10570) is “ten thousand five hundred

```python
def anglicize(n):
    """Returns: the anglicization of int n.

    Precondition: 0 < n < 1,000,000"

    pass # ???
```
def anglicize(n):
    """Returns: the anglicization of int n.

    Precondition: 0 < n < 1,000,000"

    # if < 1000, provide an answer

    # if > 1000, break into hundreds, thousands parts
    # use the < 1000 answer for each part , and glue
    # together with "thousands" in between

    return " " # empty string
Exercise: Anglicizing an Integer

```python
def anglicize(n):
    """Returns: the anglicization of int n.

    Precondition: 0 < n < 1,000,000"
""
    if n < 1000:
        # no thousands place
        return anglicize1000(n)
    elif n % 1000 == 0:
        # no hundreds, only thousands
        return anglicize1000(n/1000) + ' thousand'
    else:
        # mix the two
        return (anglicize1000(n/1000) + ' thousand ' +
                anglicize1000(n))
```
def anglicize(n):
    """Returns: the anglicization of int n.
    Precondition: 0 < n < 1,000,000"
    if n < 1000:  # no thousands place
        return anglicize1000(n)
    elif n % 1000 == 0:  # no hundreds, only thousands
        return anglicize1000(n/1000) + ' thousand'
    else:  # mix the two
        return (anglicize1000(n/1000) + ' thousand ' +
                anglicize1000(n))

Now implement this. See anglicize.py