Lecture 8

Algorithm Design
Announcements for Today

Assignment 1

- We have finished grading!
  - Resubmit until correct
- If you were close…
  - Will get feedback in CMS
  - Fix your assignment
- If you were very wrong…
  - You got an e-mail
  - Holding 1-on-1s this week
- FINISH THE SURVEY

Reading

- Read Chapter 4
- No reading for Thursday

More Assignments

- A2 due next week (Sunday)
- A3 posted on Saturday
  - Due 2 weeks from Fri
  - Before leave for Fall Break
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*
**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 cannot "understand" you
  - Python does what you say, not what you meant

9/23/14

Alogrithm Design
Stepwise Refinement: Basic Principles

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

- **Take Small Steps**
  Do a little at a time; follow the *Mañana Principle*

- **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
Mañana Principle

• If not in current step, delay to “tomorrow”
  ▪ 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
```
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
    with one blank between names""
    end = s.find(' ')
    return s[:end]
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
    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
def anglicize(n):
    """Returns: the anglicization of int n.
    Precondition: 0 < n < 1,000,000""
    if n < 1000:  # no thousands place
        return anglicize1000(n)
    if n % 1000 == 0:  # no hundreds, only thousands
        return anglicize1000(n/1000) + ' thousand'
    else:  # mix the two
        return (anglicize1000(n/1000) + ' thousand ' +
                anglicize1000(n))
Exercise: Anglicizing an Integer

def anglicize(n):
    """Returns: the anglicization of int n.
    Precondition: 0 < n < 1,000,000"

    if n < 1000:
        # no thousands place
        return anglicize1000(n)
    if 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
Helper Functions and Errors

# error.py

```python
def function_1(x,y):
    return function_2(x,y)

def function_2(x,y):
    return function_3(x,y)

def function_3(x,y):
    return x/y  # crash here

if __name__ == '__main__':
    print function_1(1,0)
```

9/23/14
Helper Functions and Errors

# error.py

```python
def function_1(x, y):
    return function_2(x, y)

def function_2(x, y):
    return function_3(x, y)

def function_3(x, y):
    return x / y  # crash here

if __name__ == '__main__':
    print function_1(1, 0)
```

# error.py

def function_1(x,y):
    return function_2(x,y)

def function_2(x,y):
    return function_3(x,y)

def function_3(x,y):
    return x/y  # crash here

if __name__ == '__main__':
    print function_1(1,0)

Traceback (most recent call last):
  File "error.py", line 20, in <module>
    print function_1(1,0)
  File "error.py", line 8, in function_1
    return function_2(x,y)
  File "error.py", line 12, in function_2
    return function_3(x,y)
  File "error.py", line 16, in function_3
    return x/y

Error list provides:
- Function where error is found
- Every function that called it
# error.py

```python
def function_1(x,y):
    return function_2(x,y)

def function_2(x,y):
    return function_3(x,y)

def function_3(x,y):
    return x/y  # crash here

if __name__ == '__main__':
    print function_1(1,0)
```

Error list provides:
- Function where error is found
- Every function that called it

Traceback (most recent call last):
File "error.py", line 20, in <module>
    print function_1(1,0)
File "error.py", line 8, in function_1
    return function_2(x,y)
File "error.py", line 12, in function_2
    return function_3(x,y)
File "error.py", line 16, in function_3
    return x/y
Helper Functions and Errors

```
# error.py

def function_1(x,y):
    return function_2(x,y)

def function_2(x,y):
    return function_3(x,y)

def function_3(x,y):
    return x/y  # crash here

if __name__ == '__main__':
    print function_1(1,0)
```

Error list provides:

- Function where error is found
- Every function that called it

The Call Stack

Motivation for next Lecture

9/23/14