Lecture 8

Algorithm Design
## Announcements For This Lecture

### Assignment 1
- **Due TOMORROW**
  - Due *before* midnight
  - Submit something…
  - Last revision Oct. 2
- 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
# What Are Algorithms?

## Algorithm

- Step-by-step instructions
  - Not specific to a language
  - Could be a cooking recipe
- **Outline** for a program

## Implementation

- Program for an algorithm
  - In a specific language
  - What we often call coding
- The **filled in** outline

- Good programmers can separate the two
  - Work on the algorithm first
  - Implement in language second
- Why approach strings as **search-cut-glue**

---

9/24/19 Algorithm Design 3
# Difficulties With Programming

## Syntax Errors

- Python can’t understand you
- **Examples:**
  - Forgetting a colon
  - Not closing a parens
- Common with beginners
  - But can quickly train out

## Conceptual Errors

- Does what you say, not mean
- **Examples:**
  - Forgot last char in slice
  - Used the wrong argument
- Happens to everyone
  - Large part of CS training

Proper algorithm design reduces conceptual errors
Testing First Strategy

- **Write the Tests First**
  Could be script or written by hand

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

- **Intersperse Programming and Testing**
  When you finish a step, test it immediately

- **Separate Concerns**
  Do not move to a new step until current is done
Testing First Strategy

- **Write the Tests First**
  Could be script or written by hand

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

- **Intersperse Programming and Testing**
  When you finish a step, test it immediately

- **Separate Concerns**
  Do not move to a new step until current is done

Will see several strategies. But all built on this core idea.
Using Placeholders in Design

- **Strategy**: fill in definition a little at a time
- We start with a function *stub*
  - Function that can be called but is unfinished
  - Allows us to test while still working (later)
- All stubs must have a function header
  - But the definition body might be “empty”
  - Certainly is when you get started
def last_name_first(s):
    """Returns: copy of s in form 'last-name, 'first-name'
    Precondition: s is in form 'first-name last-name'
    with one blank between the two names"""
    # Finish the body
def last_name_first(s):
    # Finish the body

• A function definition is only valid with a body
  ▪ (Single-line) comments do not count as body
  ▪ But doc-strings do count (part of help function)

• So you should always write in the specification
An Alternative: Pass

def last_name_first(s):
    pass

• You can make the body non-empty with pass
  ▪ It is a command to “do nothing”
  ▪ Only purpose is to ensure there is a body

• You would remove it once you got started
def last_name_first(s):

    """Returns: copy of s in form 'last-name, first-name'
    Precondition: s is in form 'first-name last-name'
    with one blank between the two names"""

    pass

Now pass is a note that is unfinished. Can leave it there until work is done.
Outlining Your Approach

- Recall the two types of errors you will have
  - Syntax Errors: Python can’t understand you
  - Conceptual Errors: Does what you say, not mean
- To remove conceptual errors, plan before code
  - Create outline of the steps to carry out
  - Write in this outline as comments
- This outline is called pseudocode
  - English statements of what to do
  - But corresponds to something simple in Python
def last_name_first(s):
    """Returns: copy of s in form 'last-name, first-name'
    Precondition: s is in form 'first-name last-name'
    with one blank between the two names"""

    # Find the space between the two names
    # Cut out the first name
    # Cut out the last name
    # Glue them together with a comma
Example: Reordering a String

def last_name_first(s):
    """Returns: copy of s in form 'last-name, 'first-name'
    Precondition: s is in form 'first-name last-name'
    with one blank between the two names""
    end_first = s.find(s,' ')
    # Cut out the first name
    # Cut out the last name
    # Glue them together with a comma
def last_name_first(s):
    """Returns: copy of s in form 'last-name, 'first-name'
    Precondition: s is in form 'first-name last-name'
    with one blank between the two names""
    end_first = s.find(s,' ') 
    first_name = s[:end_first] 
    # Cut out the last name 
    # Glue them together with a comma
What is the Challenge?

• Pseudocode must correspond to Python
  ▪ Preferably implementable in one line
  ▪ **Unhelpful**: # Return the correct answer

• So what can we do?
  ▪ Depends on the types involved
  ▪ Different types have different operations
  ▪ You should memorize important operations
  ▪ Use these as **building blocks**
Case Study: Strings

• We can **slice** strings (s[a:b])
• We can **glue** together strings (+)
• We have a lot of string **methods**
  ▪ We can **search** for characters
  ▪ We can **count** the number of characters
  ▪ We can **pad** strings
  ▪ We can **strip** padding
• Sometimes, we can **cast** to a new type
Early Testing

• **Recall**: Combine programming & testing
  - After each step we should test
  - But it is unfinished; answer is incorrect!

• **Goal**: ensure *intermediate results* expected
  - Take an input from your testing plan
  - Call the function on that input
  - Look at the results at each step
  - Make sure they are what you expect

• Add a **temporary return value**
def last_name_first(s):
    """Returns: copy of s in form 'last-name, 'first-name'
    Precondition: s is in form 'first-name last-name'
    with one blank between the two names"""
    end_first = introcs.find_str(s, ' ')  
    first = s[:end_first]  
    # Cut out the last name
    # Glue them together with a comma
    return first  
    # Not the final answer
Working with Helpers

• Suppose you are unsure of a step
  ▪ You maybe have an idea for pseudocode
  ▪ But not sure if it easily converts to Python

• But you can specify what you want
  ▪ Specification means a new function!
  ▪ Create a specification stub for that function
  ▪ Put a call to it in the original function

• Now can lazily implement that function
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""

    # Cut out the first name
    # Cut out the last name
    # Glue together with comma
    # Return the result
Example: last_name_first

```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)

    # Cut out the last name
    # Glue 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""
    pass
```
Example: last_name_first

```python
def last_name_first(s):
    
    """Returns: copy of s in the form
    'last-name, first-name'
    """

    first = first_name(s)

    # Cut out the last name
    # Glue together with comma
    return first # Stub

    
    """Precondition: s is in the form
    'first-name last-name' with
    one blank between names"
    
    # Cut out the last name
    # Glue 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]
```
Concept of Top Down Design

• Function specification is **given** to you
  ▪ This cannot change at all
  ▪ Otherwise, you break the team
• But you **break it up** into little problems
  ▪ Each naturally its own function
  ▪ **YOU** design the specification for each
  ▪ Implement and test each one
• Complete before the main function
def test_first_name():
    """Test procedure for first_name(n)""
    result = name.first_name('Walker White')
    introcs.assert_equals('Walker', result)

def test_last_name_first():
    """Test procedure for last_name_first(n)""
    result = name.last_name_first('Walker White')
    introcs.assert_equals('White, Walker', result)
A Word of Warning

• **Do not go overboard** with this technique
  - Do not want a lot of one line functions
  - Can make code harder to read in extreme

• **Do it if the code is too long**
  - I personally have a one page rule
  - If more than that, turn part into a function

• **Do it if you are repeating yourself a lot**
  - If you see the same code over and over
  - Replace that code with a single function call
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 the result
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