Module 10

Algorithm Design

Focus of this Video Series

- You know how to write a function definition
    - Have shown you the basic definition syntax
    - Have shown you what happens on a call
- But different that implementing a function
    - Given an English description of what to do
    - You have to write code that meets spec
    - This is the real skill that earns people money
- How to do that is focus of this series

Starting with the Specification

```
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
    """

    # Finish the body
```

Analogy: Math word problems

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**

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 **ers**

- **Inte**

  Whe st 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.

The Role of Stubs

- **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

A Function Stub

```
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
```

“Empty”

But it Cannot Really Be Empty

```python
def last_name_first(s):
    # Finish the body
```

Error

- 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
```

Fine!

- 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

Ideally: Use Both

```
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

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"""
    # Find the space between the two names
    # Get the first name
    # Get the last name
    # Put 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(' ')
    # Get the first name
    # Get the last name
    # Put 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(' ')
    first = s[:end_first]
    # Get the last name
    # Put 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 features in `introcs`
  - 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

Working With an Unfinished Function

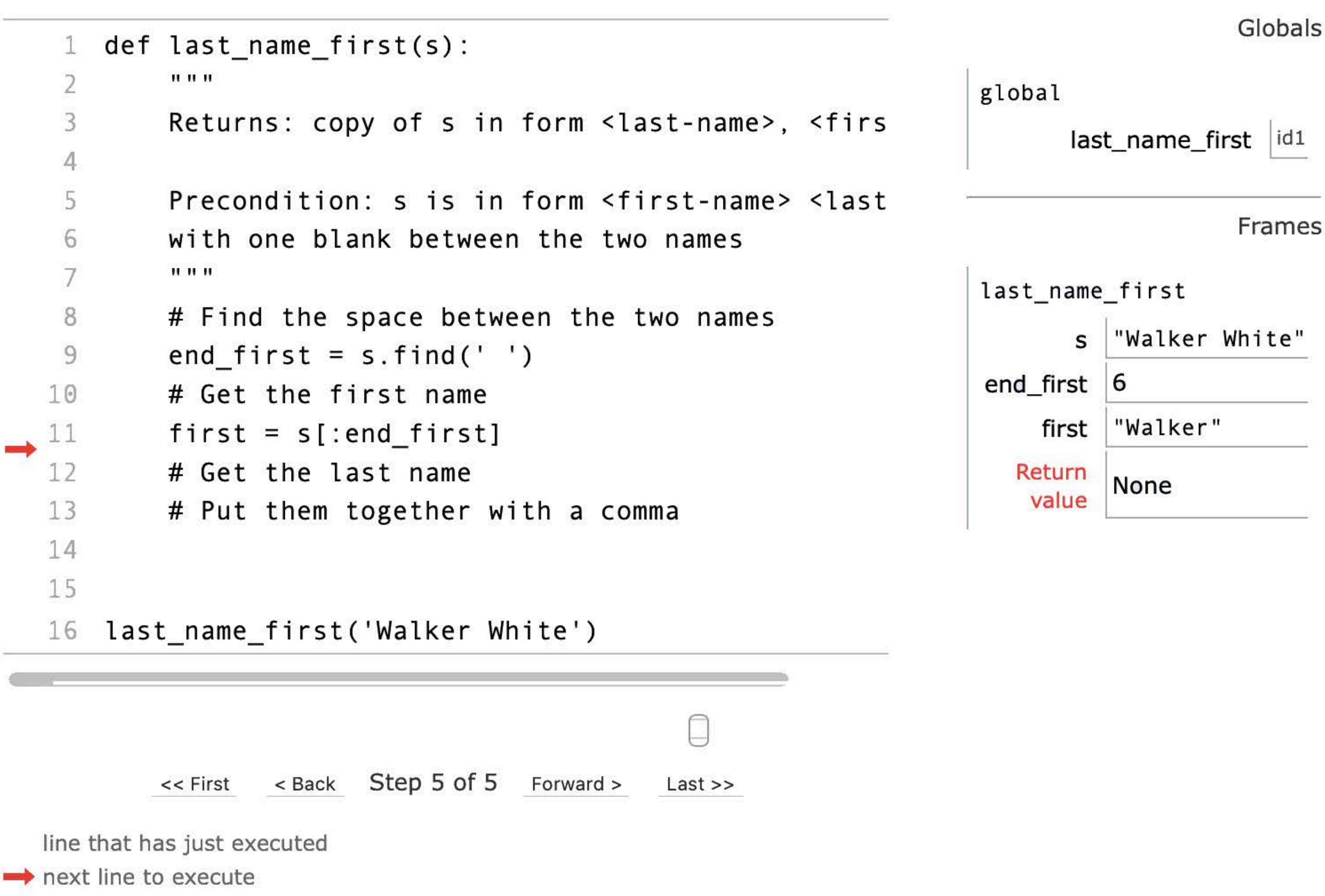
```
def last_name_first(s):
    """
    Returns: copy of s in form <last-name>, <first-name>

    Precondition: s is in form <first-name> <last-name>
    with one blank between the two names"""
    end_first = s.find(' ')
    first = s[:end_first]
    # Get the last name
    # Put them together with a comma
```

How do we test this code?

Early Testing

- **Recall**: Intersperse 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
- This requires the Python Tutor

Visualizing with the Python Tutor



```python
def last_name_first(s):
    """
    Returns: copy of s in form <last-name>, <firs

    Precondition: s is in form <first-name> <last
    with one blank between the two names
    """
    # Find the space between the two names
    end_first = s.find(' ')
    # Get the first name
    first = s[:end_first]
    # Get the last name
    # Put them together with a comma


last_name_first('Walker White')
```

Globals

global

last_name_first	id1

Frames

last_name_first

s	"Walker White"
end_first	6
first	"Walker"
Return value	None

line that has just executed

→ next line to execute

Alternative: Print Statements

- Don’t always have the Python Tutor
  - Python Tutor is not full featured
  - Sometimes must test directly with Python
- Could use **print statements** to see
  - We did this when debugging
  - Principle is the same here
  - But remember to remove these
  - …or at least comment out

Alternative: Stubbed Returns

- **Idea**: We can always see a return value
  - Assume calling in the interactive shell
  - Return is the evaluation of the call
- Add a return statement to end of function
  - Return the variable we want to visualize
  - Different from the eventual return expression
  - Why we call it a stubbed return

Alternative: Stubbed Returns

```
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(' ')
    first = s[:end_first]
    # Get the last name
    # Put them together with a comma
    return first        # Not the final answer
```

Rethinking the Backwards Approach

- The advantage of backwards approach?
  - You could be “lazy” in the design
  - If you were not sure, make it a variable
  - Define that variable in a previous line
- What if we could do it forwards?
  - Still have this lazy design approach
  - But now could do incremental testing
  - Seems best of both worlds

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 clearly 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

Example: last_name_first

```
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"""
    # Find the first name

    # Find the last name

    # Put together with comma

    return first # Stub
```

Example: last_name_first

```
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
```

```
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
```

Example: last_name_first

```
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]
```

Concept of Top Down Design

- Function pecification 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

Testing and Top Down Design

```
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

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

    Precondition: 0 < n < 1,000,000"""
    pass # ???
```

- We first step through some examples
  - Like coming up with the test cases
  - But we also look for patterns in the answers
- From these patterns, we break into cases
  - And we combine with top-down design

Stepping Through Examples

- **Examples:**
  - 3 => "three"
  - 53 => "fifty three"
  - 253 => "two hundred fifty three"
  - 3253 => "three thousand two hundred fifty three"
  - 253253 => "two hundred fifty three thousand
    two hundred fifty three"
- Already see a pattern
  - Rules for each group of three numbers are same

Approaching with Top Down Design

```
def anglicize(n):
    """Returns: the anglic

    Precondition: 0 < n <
    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

Moving on to the Next Function

```
def anglicize1000(n):
    """Returns: the anglicization of int n.

    Precondition: 0 < n < 1,000"""
    pass # ???
```

- Notice it is essentially same problem as before
  - ONLY thing changed is the precondition
  - So it limits the number of cases to look at
- But we want to break it up further
  - Want to handles 1, 2, and 3 digit separately

More Top Down Design

```
def anglicize1000(n):
    """Returns: the anglicization of int n.

     Precondition: 0 < n < 1,000"""
     # Determine number of "dig
     if n < 20:
         return anglicize1to19(n)
     elif n < 100:
         return anglicize20to99(n)
     else:
         return anglicize100to999(n)
```

Must Brute Force

Needs a tens helper

Now straightforward

See Module bugs.py

```
def valid_date(date):
    """Returns: True if date is an actual date

    Example: valid_date('2/29/2004') is True
    but valid_date('2/29/2003') is False

    Precond: date is a string month/day/year where
    month, day are 1 or 2 digit each and year is 4"""
    # Split up string
```

Bug Number 1

```
>>> valid_date('3/30/2004')
First / at 1
Second / at 4
Month is 3
Day is 30
Year is 3
Leap year
Month is February
Month 3
    has 29 days
Day out of range
False
```

- **Note:** Weird trace
  - Month is February
  - Tells us what is wrong
- Change line 98

```
elif (month == 2 and
        leap_year(year)):
print('Month is February')
```

Bug Number 2

```
>>> valid_date('2/2/2000')
First / at 1
Second / at -1
Month is 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "bugs.py", line 33, in valid_date
    day   = int(date[pos1+1:pos2])
ValueError: invalid literal for int()
with base 10: '2/200'
```

- **Note:** Search failed
  - Could not find /
  - Tells us what is wrong
- Change line 32

```
pos2 =
date.find('/',pos1+1)
```