Module 16

For-Loops
Motivating Example

```python
def print_each(text):
    """Prints each character of text on a line by itself

Example: print_each('abc') displays
    a
    b
    c

Parameter text: The string to split up
Precondition: text is a string""
```
def print_each(text):
    """Prints each character of text on a line by itself

    Precondition: text is a string """
    print(text[0])
    print(text[1])
    ...
    print(text[len(text)-1])

Unfortunately not valid Python
The Problem

- **Strings are potentially unbounded**
  - Number of characters inside them is not fixed
  - Functions must handle different lengths
  - **Example**: `print_each('a') vs. print_each('abcdfgh')`

- Cannot process with **fixed** number of lines
  - Each line of code can handle at most one element
  - What if # of elements > # of lines of code?

- We need a new **control structure**
The For-Loop

# Create local var x

x = text[0]
print(x)

x = text[1]
print(x)

...

x = text[len(text)-1]
print(x)

# Write as a for-loop

for x in text:
    print(x)

Key Concepts

- **iterable**: text
- **loop variable**: x
- **body**: print(x)
The For-Loop

# Create local var x

```python
x = text[0]
print(x)
```

```python
x = text[1]
print(x)
```

```python
...
```

```python
x = text[len(text)-1]
print(x)
```

# Write as a for-loop

```python
for x in text:
    print(x)
```

• **Iterable can be a string, tuple or list**

### Key Concepts

- **iterable**: `text`
- **loop variable**: `x`
- **body**: `print(x)`
Executing a For-Loop

The for-loop:

```python
for x in text:
    print(x)
```

- iterable: `text`
- loop variable: `x`
- body: `print(x)`
def print_each(text):
    """Prints each char of text
    Pre: text is a string"""
    for x in thelist:
        print(x)

print_each(word):
def print_each(text):
    
    """Prints each char of text
    Pre: text is a string"""

    for x in thelist:
        print(x)

print_each(word):
def print_each(text):
    """Prints each char of text
    Pre: text is a string"""
    for x in thelist:
        print(x)

print_each(word)

Loop back to line 4
def print_each(text):
    """Prints each char of text
    Pre: text is a string"""
    for x in thelist:
        print(x)

print_each(word):

Next element stored in x.
Previous value is lost.
def print_each(text):
    """Prints each char of text
    Pre: text is a string""
    for x in thelist:
        print(x)
    print_each(word)
def print_each(text):
    """Prints each char of text
    Pre: text is a string"""
    for x in thelist:
        print(x)

print_each(word):

Loop is completed.
Nothing new put in x.
def print_each(text):
    
    """Prints each char of text
    Pre: text is a string"""

    for x in thelist:
        print(x)
Example: **Summing Elements of a Tuple**

```python
def sum(tups):
    """Returns: the sum of all elements in tups
    Precondition: tups is a tuple of all numbers
    (either floats or ints)"
    pass  # Stub to be implemented
```

Remember our approach:
Outline first; then implement
Example: Summing Elements of a Tuple

def sum(tups):
    """Returns: the sum of all elements in tups
    Precondition: tups is a tuple of all numbers (either floats or ints)"
    # Create a variable to hold result (start at 0)
    # Add each tuple element to variable
    # Return the variable
Example: Summing Elements of a Tuple

```python
def sum(tups):
    
    """Returns: the sum of all elements in tups
    Precondition: tups is a tuple of all numbers (either floats or ints)"

    result = 0

    for x in tups:
        result = result + x

    return result
```

- **iterable:** `tups`
- **loop variable:** `x`
- **body:** `result = result + x`
def num_ints(tups):

    """Returns: the number of ints in tups
    Precondition: tups is a tuple of any mix of types"
    
    # Create a variable to hold result (start at 0)
    # for each element in the tuple...
        # check if it is an int
        # add 1 if it is
    
    # Return the variable

def num_ints(tups):

    """Returns: the number of ints in tups
    Precondition: tups is a tuple of any mix of types"

    result = 0

    for x in tups:
        if type(x) == int:
            result = result + 1

    return result
The Accumulator

• In a previous example saw the **accumulator**
  ▪ Variable to hold a final (numeric) answer
  ▪ For-loop added to variable at each step
• This is a common *design pattern*
  ▪ Popular way to compute statistics
  ▪ Counting, averaging, etc.
• It is not just limited to numbers
  ▪ Works on every type that can be *added*
  ▪ This means strings, lists and tuples!
def despace(s):
    """Returns: s but with its spaces removed
    Precondition: s is a string"""
    # Create an empty string accumulator
    # For each character x of s
    # Check if x is a space
    # Add it to accumulator if not
**Example: String-Based Accumulator**

```python
def despace(s):
    """Returns: s but with its spaces removed
    Precondition: s is a string"""
    result = ''
    for x in s:
        if x != '':
            result = result+x
    return result
```
def reverse(s):
    """Returns: copy with s with characters reversed.
    Example: reverse('hello') returns 'olleh'
    Precondition: s is a (possibly empty string)"""
    # Create an empty tuple accumulator
    # For each character x of s
    # Add x to FRONT of accumulator
def reverse(s):
    
    """Returns: copy with s with characters reversed.
    Example: reverse('hello') returns 'olleh'
    Precondition: s is a (possibly empty string)"
    
    result = ''
    for x in s:
        result = x + result
    return result
Example: List-Based Accumulator

def copy_add_one(lst):
    """Returns: copy with 1 added to every element
    Precondition: lst is a list of all numbers
    (either floats or ints)"""
    # Create an empty tuple accumulator
    # For each element x of lst
    # Add 1 to value of x
    # Add x to the accumulator
def copy_add_one(lst):
    """Returns: copy with 1 added to every element
    Precondition: lst is a list of all numbers
    (either floats or ints)""
    copy = []  # accumulator
    for x in lst:
        x = x + 1
        copy = copy + [x]
    return copy
def copy_add_one(lst):
    """Returns: copy with 1 added to every element
    Precondition: lst is a list of all numbers (either floats or ints)"

    copy = []  # accumulator
    for x in lst:
        x = x + 1
        copy.append(x)  # add to end of copy
    return copy
The Comparison

• They appear to be the same
• But first is less efficient (TURN ARROWS OFF)

• List accums are preferable for large data
def hello(n):
    """Prints 'Hello World' n times
    Precondition: n > 0 is an int."""
    pass # Stub to be implemented
def hello(n):
    """Prints 'Hello World' n times
    Precondition: n > 0 is an int."""
    lst = [1, 2, ..., n]
    for x in lst:
        print('Hello World')

How do we do this step?
# The Range Iterable

**range**(x)

- Creates an *iterable*
  - Can be used in a for-loop
  - Makes ints (0, 1, ... x-1)
- But it is not a tuple!
  - A **black-box** for numbers
  - Entirely used in for-loop
  - Contents of folder hidden

## Example

```python
>>> range(3)
range(0,3)
>>> for x in range(3)
...       print(x)
0
1
2
```
def hello(n):
    """Prints 'Hello World' n times
    Precondition: n > 0 is an int."""
    for x in range(n):
        print('Hello World')
Uses of Range

• Can convert to list
  ▪ Remember: iterable!
  >>> list(range(4))
  [0, 1, 2, 3]

• Best for handling ints
  ▪ Statistical calculations
  ▪ Computing n samples

• Or fixed repeats

```python
def sum_squares(n):
    """
    Rets: sum of squares to n
    Prec: n is int > 0
    """
    total = 0
    for x in range(n):
        total = total + x*x
```

Accumulator
Two Main Variations

- **range(a,b)**
  - Generates \((a,\ldots,b-1)\)
  - Useful when do not want to start at 0
  - Requires that \(b > a\)

- **range(a,b,n)**
  - Generates \((a,a+n,\ldots,b-1)\)
  - “Counting by evens (or threes)”
  - \(n\) must be \(> 0\)
def partition(s):

    """Returns: a list splitting s in two parts

    The 1st element of the tuple is chars in even positions (starting at 0), while the 2nd is odds.

    Examples:
    partition('abcde') is ['ace','bd']
    partition('aabb') is ['ab', 'ab']

    Precondition: s is a string.""

pass  # Stub to be implemented
def partition(s):
    """Returns: a list splitting s in two parts
    Precondition: s is a string."""
    # Create accumulators for first & second parts
    # For each character in s
        # Determine if character is at odd or even pos
        # Add it to the correct accumulator
    # Return list with the two parts
def partition(s):
    
    """Returns: a list splitting s in two parts
    Precondition: s is a string."""

    first = ''; second = ''
    for x in s:
        pos = s.find(x)
        if pos % 2 == 0:
            first = first + x
        else:
            second = second + x

    return [first,second]
Getting Positions

• We want the positions!
  ▪ So loop over the positions, not elements
  ▪ If have position, can access with $s[pos]$

• Notice that range(n) starts at 0
  ▪ This is first position of a string/list/tuple

\[
\text{lst} = [5, 2, 7, 1]
\]
\[
\text{pos} = [0, 1, 2, 3]
\]

• So use \text{range}(\text{len}(\text{lst}))
def partition(s):
    """Returns: a list splitting s in two parts
    Precondition: s is a string."""
    first = ''
    second = ''
    for pos in range(len(s)):
        if pos % 2 == 0:
            first = first + s[pos]
        else:
            second = second + s[pos]
    return [first,second]
**Motivation: A Mutable Function**

```python
def add_one(lst):
    """(Procedure) Adds 1 to every element in the list
    Precondition: lst is a list of all numbers
    (either floats or ints)""
```

- Accumulator pattern no longer relevant
  - Do not want to accumulate a new list
  - Want to modify the original list
- What is the right way to approach this?
A Motivating Function

def add_one(lst):
    """(Procedure) Adds 1 to every element in the list
    Precondition: lst is a list of all numbers
    (either floats or ints)"""
    for x in lst:
        x = x + 1
    # procedure; no return

We need to put the answer into lst
Modifying a Loop Variable is Unsafe!

- This is an infinite loop:

```python
for x in lst:
    lst.append(1)
```

- Best practices?
  - Never modify a loop var
  - Pick another iterable
  - Use that to modify first
def add_one(lst):
    """(Procedure) Adds 1 to every element in the list
    Precondition: lst is a list of all numbers (either floats or ints)""

    size = len(lst)
    for k in range(size):
        lst[k] = lst[k] + 1
    # procedure; no return
Testing For-Loops

• Once again, we need code coverage
• But is automatic from **Rule of Numbers**
  - **Rule of 1:** Executes loop just once
  - **Rule of 2:** Executes loop many times
  - **Rule of 0:** Skips over loop entirely
• The hard part is what to do about **lists**
  - What if function is a mutable procedure?
  - What is the function is *accidentally* mutable?
• How do we have to adapt the test scripts?
Testing Immutable For-Loop

def copy_add_one(lst):
    """Returns: copy with 1 added to every element
    Precondition: lst is a list of all numbers
    (either floats or ints)"
    ...

x = [1,2]
result = copy_add_one(x)
introcs.assert_equals([2,3], result)
introcs.assert_equals([1,2], x)

Verify the output (the return value)

Check that it is not accidentally mutable
def add_one(lst):
    """(Procedure) Adds 1 to every element in the list
    Precondition: lst is a list of all numbers (either floats or ints)"""

... x = [1,2]
result = add_one(x)
introcs.assert_equals([2,3],x)
introcs.assert_equals(None,result)

Verify the output (modified argument)
Check that it is not accidentally fruitful
Tuple Expansion

• Last use of lists/tuples is an advanced topic
  ▪ But will see if read Python code online
  ▪ Favored tool for data processing

• An Observation:
  ▪ Function calls look like name + tuple
  ▪ Why not pass a *single* argument: the tuple?

• Purpose of tuple expansion: *tuple*
  ▪ But only works in certain contexts
Tuple Expansion Example

```python
>>> def add(x, y)
...     '''Returns x+y'''
...     return x+y
... >>> a = (1,2)
... >>> add(*a)    # Slots each element of a into params
... 3
... >>> a = (1,2,3) # Sizes much match up
... >>> add(*a)
... ERROR
```

Have to use in function call

# Slots each element of a into params
3

# Sizes much match up
ERROR
Also Works in Function Definition

```python
1  def max(*tup):
2      themax = None
3      for x in tup:
4          if themax is None or themax < x:
5              themax = x
6      return themax
7
8
9  a = max(1,2)
10  b = max(1,2,3)
```

Line that has just executed:

- line 10

Next line to execute:

- line 2
def max(*tup):
    """Returns the maximum element in tup
    Param tup: The tuple of numbers
    Precond: Each element of tup is an int or float"
    themax = None
    for x in tup:
        if themax == None or themax < x:
            themax = x
    return themax