Module 6: More Python features

Topics:

- Printing to standard output
- Strings and their methods
- Reading from standard input
- Lists and their methods
- Abstract list functions

Readings: ThinkP 8, 10

Run the following program in the Definitions window. What do you see?

```
def middle(a,b,c):
    largest = max(a,b,c)
    smallest = min(a,b,c)
    mid = (a+b+c) - largest - \
          smallest
    return mid
middle(10,20,30)
middle(0,10,-10)
middle(-1,-3,-2)
```

Python output: printing information to the screen

```
x = 20
print x
print x+5
y = "dog"
print y
z = 42.8
print z
print x, y, z
```

More on `print`

- Does not produce a value, but has an effect
  - The Effects section of a function **must** describe any information that is printed by the function (Design Recipe)
  - Use parameter names in your description
- The following statements are not valid Python:

```
x = print 42.8
print (x = 4)
```

Why?

Displaying values in Python programs

- Interactions window, for variable **`x`**:

```
x
print x
```

- Result *usually* looks the same (except for strings), but are different
- Difference is obvious in Definitions window

➔ Need to use **`print`** in our programs to see results as the program is running

Example: Write a function that prints a string three times – once per line

```
# print_it_three_times: str -> None
# Purpose: produces None
# Effects: Prints the string s three times,
# once per line
# Example: print_it_three_times("a") prints
#a
#a
#a
def print_it_three_times(s):
    print s
    print s
    print s
```

Testing Screen Output

- Give a description of expected screen output:

```
check.set_screen(
     "CS 116 on three lines")
```

- Call appropriate **check** function to test value produced by the function (even if it is None)
- Test will print screen output along with your description of what the screen output should be
- You must then compare the two.

Example: Screen Output Only

```
import check
def print_it_three_times(s):
    print s
    print s
    print s

# Q6 Test 1: a short string - "CS 116"
check.set_screen("CS 116 on three lines")
check.expect("Q6T1",
    print_it_three_times("CS 116"),None)
```

There is no **return**, so function produces **None**. This value is passed to **check.expect** to verify.

Test Output

```
QT1 (expected screen output):
CS 116 on three lines

QT1 (actual screen output):
CS 116
CS 116
CS 116
-----
```

You must examine your output to see if it matches what you expected.

Note: No error message printed by **check.expect**, so None was correctly returned by our function.

Printing vs Returning

In Scheme, most of our functions produced a value. This will not be the case in Python.

Complete the design recipe for **f1** and **f2**.

```
def f1(x):
   print x+1
def f2(x):
   return x+1
```

Debugging your program with `print` statements

- If you have an error in your program, place **print** statements at points through out your program to display values of variables
- **IMPORTANT**: Remember to remove the **print** statements before submitting your code.
  - Your program may fail our tests, even if it produces the correct function values!!!

Strings in Python: combining strings in interesting ways

```
s = "Great"
t = "CS116"
print s + t
print s + "!!!! " + t
print s * 3, 2 * t
print 'single quote works too'
print 'strings can contain
 quotes" too'
```

Overloading of *

The following are all valid contracts of `*`:

```
*: int int -> int
*: int float -> float
*: float int -> float
*: float float -> float
*: int str -> str
*: str int -> str
```

Other string operations

- Contains substring: **`s in t`**
  - Produces **`True`** if the string `s` appears as a substring in the string **`t`**

```
"astro" in "catastrophe" => True
"car" in "catastrophe" => False
"" in "catastrophe" => True
```

- String length: **`len(s)`**
  - Produces the number of characters in string **`s`**

```
len("") => 0,
len("Billy goats gruff!") => 18
```

Extracting substrings

- **`s[i:j]`** produces the substring from string `s`, containing all the characters in positions **`i, i+1, i+2, ..., j-1`**
- Like Scheme, strings in Python start from position 0

```
s = “abcde”
print s[2:4]
print s[0:5]
print s[2:3]
print s[3:3]
print s[2:]
print s[:3]
print s[4]
```

Strings are immutable

We cannot change the individual characters in a string **`s`**

`s = "abcde"`

`s[3] = "X"` causes an error

but

`s = s[:3] + "X" + s[4:]`

produces a new string **`"abcXe"`** and assigns it to **`s`**

Methods in Python

- **`str`** is name of the string type in Python
- It is the also the name of a module in Python
- Like the **`math`** module, **`str`** contains many functions to process strings
- To use the functions in **`str`**:

```
s = "hi"
str.upper(s) => "HI"
```

- Even easier – use special dot notation:

```
s.upper() => "HI"
```

- Note that none of the string methods modify the string itself

Full listing of string methods

```
>>> dir("abc")
['__add__', '__class__', '__contains__', '__delattr__',
   '__doc__', '__eq__', '__format__', '__ge__',
   '__getattribute__', '__getitem__', '__getnewargs__',
   '__getslice__', '__gt__', '__hash__', '__init__', '__le__',
   '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
   '__new__', '__reduce__', '__reduce_ex__', '__repr__',
   '__rmod__', '__rmul__', '__setattr__', '__sizeof__',
   '__str__', '__subclasshook__',
   '_formatter_field_name_split', '_formatter_parser',
   'capitalize', 'center', 'count', 'decode', 'encode',
   'endswith', 'expandtabs', 'find', 'format', 'index',
   'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',
   'istitle', 'isupper', 'join', 'ljust', 'lower',
   'lstrip', 'partition', 'replace', 'rfind', 'rindex',
   'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
   'splitlines', 'startswith', 'strip', 'swapcase',
   'title', 'translate', 'upper', 'zfill']
```

Using string methods

```
s = 'abcde 1 2   3 ab    '
>>> s.find('a')
>>> s.find('a',1)
>>> s.split()
>>> s.split('a')
>>> s.startswith('abc')
>>> s.endswith('b')
```

Getting more information about a **string** function

```
>>> print "".isalpha.__doc__
S.isalpha() -> bool

Return True if all characters in S
  are alphabetic and there is at
  least one character in S, False
  otherwise.
```

Exercise

Write a Python function that consumes a non-empty first name, middle name (which might be empty), and a non-empty last name, and constructs a userid consisting of first letter of the first name, first letter of the middle name, and the last name. The userid must be in lower case, and no longer than 8 characters, so truncate the last name if necessary.

For example, **`userid("Harry", "James", "Potter") => "hjpotter"`**

A new Python feature

- Python functions can use information received in three different ways –
  - Two ways we have seen in Scheme:
    - Parameters
    - Global constants
  - A new way:
    - Entered via the keyboard

User Input to a Python Program

```
user_input = raw_input()
```

- Program stops
- Nothing happens until the user types at keyboard
- When user hits return, a string containing all the characters before the return is produced by `raw_input`
- The string value is used to initialize the variable `user_input`
- Program continues with new value of `user_input`

More on user input

- Alternate form (preferred):

```
user_input = raw_input(prompt)
```

e.g.

```
city = raw_input("Enter hometown:")
```

- Prints the value of `prompt` before reading any characters
- Value produced by `raw_input` is **always** a string

User Input and the Design Recipe

- When a function includes a **`raw_input`** call, this must be described in the Effects section of the Design Recipe
  - Describe what happens with the value entered by the user
  - Use parameter names in your description, if relevant

A Simple Program using **`raw_input`**: Design Recipe steps

```
# repeat_str: None -> None
# Purpose: Produces None
# Effects: The user enters a string,s, and a
#   number, n, when prompted, and prints the
#   string containing n copies of s
# Example: if the user enters "abc" and 4
#   when repeat_str() is called,
#   "abcabcabcabc" is printed
# If the user enters "" and 100 when
#   repeat_str is called, "" is printed
```

A Simple Program using **`raw_input`**

```
def repeat_str():
  s = raw_input("Enter string: ")
  t = raw_input("Enter int>=0: ")
  n = int(t)
  print n*s
```

Testing With User Input

- Set the user inputs needed for the test in order
- Always use strings for the input values

```
check.set_input(["CS116","3"])
```

- Call appropriate **check** function for produced value of function
- Test function will automatically use these values (in order) when a value is expected from **raw_input**
- You will be warned if the list contains too few or too many values

Example: Test with User Input

```
import check

def add_two_inputs():
    x = int(raw_input("Enter 1nd integer: "))
    y = int(raw_input("Enter 2nd integer: "))
    return x+y

# Test 1: two positive numbers
check.set_input(["2","7"])
check.expect("AddT1", add_two_inputs(), 9)
```

Example

Write the Python function **n_times** that reads an integer **n** from the user via the keyboard, and prints out **n** once per line on **n** lines.

More on strings:
Formatting screen output

- We can print strings

```
print “my dog has fleas”
```

- We can print integers

```
fleacount = 12
print fleacount
```

- We can even combine them

```
print ‘my dog has’, fleacount, \
      ‘fleas’
print ‘my dog has ’ +  \
  str(fleacount) + ‘ fleas’
```

Creating formatted strings: %

The format operator %

- We can describe the string we want to build, indicating where values should be inserted
- Then supply the values to insert

```
fleastring = ‘My dog has %d fleas’
  % fleacount
print fleastring
```

`description % fields`

- **`description`**
  - The string you are building
  - Uses `%` inside to show where a value should be inserted in the new string
    - **`%d`** – insert an integer (alternative:**`%i`**)
    - **`%s`** – insert a string
    - **`%g`** – insert a floating point number
- **`fields`**
  - Expression for the value

We can insert multiple values!

- **`description`** can have several `%` formatters
- **`fields`** must include the same number of values to insert as **`description`**
  - **`fields`** is expressed as a tuple
    - Immutable
    - Defined with `()` brackets

Example

```
import math
A = 3.3
B = 4.5
hypotenuse = math.sqrt(A**2 + B**2)
print 'side lengths: %g, %g
  hypotenuse: %g' % (A, B, hypotenuse)

# Compare this to not using %
print 'side lengths: ' + str(A) + ', '
  + str(B) + ' hypotenuse: ' +
  str(hypotenuse)
print 'side lengths:',A, ', ',B,'
  hypotenuse: ', hypotenuse
```

Possible errors in formatting

- Incorrect number of values to insert

```
>>> print “%g %d %g” % (42.0, 12)
TypeError: not enough arguments for
  format string
>>> print “%g %d” % ( 42.0, 12, 107.2)
TypeError: not all arguments converted
  during string formatting
```

- Incorrect types of values being inserted

```
>>> print “%d %s” % (“Two”, “times”)
TypeError: %d format: a number is
  required, not str
```

More on formatting strings with floating point numbers

- `%g` is used to in the description to insert a floating point number
  - – `%g` “adapts” to the number, and doesn’t display trailing zeroes
- `%f` can also be used
  - – `%f` will always use 6 places after the decimal point, unless explicitly indicated otherwise
  - – `%.3f` will only use 3 places after the decimal

Printing on one line

- Recall that

```
print “this goes”,”on”,”one line”
print “this on the next”
print “and so on”
```

goes on three separate lines

- However,

```
print “this goes”,”on”,”one line”,
print “and this on the same”,
print “and so on”
```

all goes on one line (due to trailing comma)

Special Characters

- So, we know how to use **`print`** statements to put information on one line
- Can you use a single print statement to put information over multiple lines?
  - – Yes, but we need a special character \n

```
print "one line\nanother\nand
  another "
```

  - – Despite taking 2 characters to type, it counts as one in string length

```
len(“A\nB\nC\n”) → 6
```

Considering **userid** again

What if userid accepted a single string, such as "Harry James Potter" instead of separate strings?

```
>>> name.split()
['Harry', 'James', 'Potter']
>>> name.split('e')
['Harry Jam', 's Pott', 'r']
```

These are lists of strings – how can we use them?

Lists in Python

- Like Scheme lists, Python lists can store
  - any number of values
  - any types of values (even in one list)
- Creating lists:
  - Use square brackets to begin and end list
  - Separate elements with a comma
- Examples:

```
num_list = [4, 5, 0]
str_list = ['a', 'b']
empty_list = []
mixed_list = ['abc', 12, True, '',
              -12.4]
```

Useful Information about Python Lists

- **len(L)** => number of items in the list **L**
- **L[i]=>** item at position **i**
  - Called indexing the list
  - Produces an error if `i` is out of range
  - Positions: **0 <= i < len(L)**
  - Actual valid range: **-len(L) <= i < len(L)**
- "Slicing" a list

  L[i:j]=>[L[i],L[i+1],…,L[j-1]]

Basic Template for Recursion

```
def f(L):
  if L == []:
    # base case action
  else:
    # … L[0] …
    # … f(L[1:]) …
```

Example:

Write a recursive Python function **`build_str`** that consumes a list of strings (**`los`**), and creates and returns a new string by concatenating together all the strings in **`los`**.

Aside: The following operation also solves this problem: ***`"".join(los)`***

Other list operations

- **`range`** function
  - **`range(a,b) => [a,a+1, …, b-1]`**
  - **`range(a) => [0,1,…, a-1]`**
  - **`range(a,b,c)`** increments by **`c`** instead of **`1`**
    - **`range(10,15,3) => [10,13]`**
    - **`range(8,5,-1) => [8,7,6]`**

Other list operations

```
>>> dir(list)
[ …, 'append', 'count',
  'extend', 'index', 'insert',
  'pop', 'remove', 'reverse',
  'sort']
```

Most of these methods mutate the list, rather than produce a new list.

You’ll need to be careful using them!

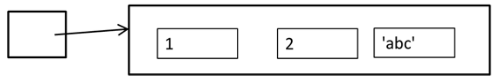
Functions vs Methods

- Methods are
  - defined in a module
  - functions that can be called in a special way

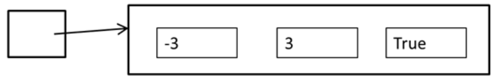
    `L.method(...)`
  - **`L`** is a parameter to **`method`**
  - **`method`** is bound to object **`L`**

Mutation and Lists

```
L = [1,2,’abc’]
```

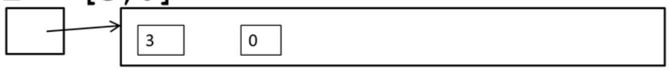


```
L[1] = 3
L[0] = -L[1]
L[2] = True
```

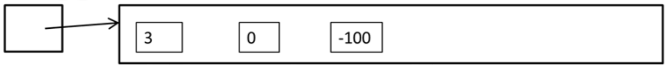


Other ways to mutate a list

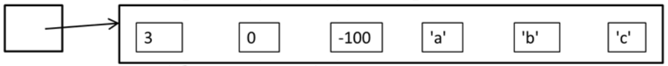
```
L = [3,0]
```



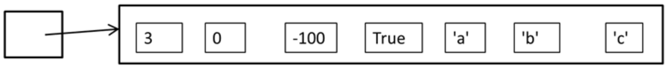
```
L.append(-100)
```



```
L.extend(['a','b','c'])
```



```
L.insert(3, True)
```



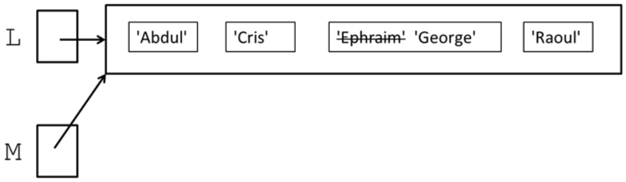
Aliasing and Lists

Recall: When two variables reference the same list, this is called *aliasing*

→ You can change the list contents using either variable name

Aliasing and Lists

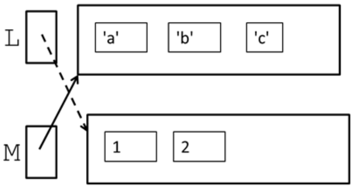
```
L=['Abdul','Cris','Ephraim','Raoul']
M = L
M[2] = 'George’
```



Breaking an Alias

As in Scheme, if we change the *value* of one variable, the other is not changed

```
L = ['a','b','c']
M = L
L = [1,2]
```



Functions and Atomic Parameters

```
def change_to_1(n):
  n = 1

grade = 89
change_to_1(grade)
print grade
```

Functions and List Parameters

```
def change_first_to_1(L):
  L[0] = 1

my_list = [‘a’, 2, ‘c’]
change_first_to_1(my_list)
print my_list
```

What is different here?

```
def change_second_to_1(L):
    L = [L[0],1] + L[2:]
    return L

my_list = [100,True,0]
print change_second_to_1(my_list)
print my_list
```

When writing a function with lists

- Important to determine if a statement in a function is supposed to
  - Use the values in an existing list,
  - Mutate an existing list, or
  - Create and return a new list
- Review ThinkP 10.12

Mutable and Immutable Values in Python

- Numbers are immutable
- Strings are immutable
- Lists are mutable
- Tuples are immutable
- Most other kinds of complicated data storage are mutable

Testing Mutation

1. Set values of state variables for testing
2. Call the appropriate **`check`** function to compare actual produced value to expected produced value (which might be **`None`**)
3. Call the appropriate **`check`** function on each testing variable that has been mutated, comparing the actual value to the expected value after mutation.

Example: Mutation

```
import check
import math

def multiply_first(L, factor):
    L[0] = L[0] * factor

## Test 1: factor = 0
L = [10,-2,3]
check.expect("T1", multiply_first(L,0), None)
check.expect("T1{L}", L, [0,-2,3])
## Test 2: factor not an integer (pi)
L = [10,0,-3.25]
check.expect("T2", multiply_first(L,math.pi), None)
check.within("T2(L[0])", L[0], 31.415926, 0.00001)
check.expect("T2(L[1])", L[1], 0)
check.within("T2(L[2])", L[2], -3.25, 0.00001)
```

Example: **`multiply_by`**

Use recursion to complete the Python function **`multiply_by`** that consumes a list of integers (**`vals`**) and another integer (**`multiplier`**) and mutates **`vals`** by multiplying each value in it by **`multiplier`**.

Lists can be used to simulate structures

```
## A posn is a list of length 2, where
## the first element is an integer or
## float (for the x coordinate), and
## and the second element is an integer
## or float for the y coordinate

## make_posn: (union int float)
##    (union int float) -> posn
def make_posn(x_coord, y_coord):
  return [x_coord, y_coord]
```

How can we implement the other **posn** functions?

```
def posn_x(p): ...
def posn_y(p): ...
def set_posn_x(p, new_x): ...
def set_posn_y(p, new_y): ...
def is_posn(v): ...
```

Other Relevant List Information

- Indexing any list element is an O(1) operation, regardless of its location in the list
- In many other languages:
  - Lists are of a fixed size once created
  - Lists can only contain one type of value
  - Processing these lists (often called arrays) tends to be faster than processing Python lists
  - Python has an `array` module (not used in CS116)

Functional Abstraction in Python: **`map`**

```
## map: (X -> Y) (listof X) ->
##      (listof Y)
## Produces a new list, applying
## function to each element in list
map(function, list)
```

```
def pull_to_passing(mark):
    if mark < 50 and mark > 46:
        return 50
    else:
        return mark
print map(pull_to_passing,
          [34, 89, 46, 49, 52])
```

Functional Abstraction in Python: **`filter`**

```
## filter: (X -> bool)
##   (listof X) -> (listof X)
## Produces a new list containing the
## elements in list for which function
## produces True
filter(function, list)
```

```
def big_enough(mark):
    return mark>50
print filter(big_enough,
          [34, 89, 46, 49, 52])
```

`lambda`

- Like Scheme, Python allows for anonymous functions using **`lambda`**
- Will be used primarily for **`map`** and **`filter`**
- Syntax:

  ```
  lambda x: expression
  lambda x,y: expression
  ```

- Note that **`expression`** cannot be a statement

What is the run-time of this function? What does it do?

```
# mystery_fn: ??? -> ???
def mystery_fn(L):
    keepers = filter(lambda s:
                  s[0]=='a', L)
    return len(keepers)

mystery_fn(['aardvark', 'A-OK',
  'cow', 'apple'])
```

Important Notes about run-time in Python

Assume list **L** contains n elements.

- **len(L)** is O(1)
- **L[index]** is O(1)
- **L+L** is O(n)
- **L[first, last]** is O(**last** - **first**)
- **filter** and **map** are at least O(n)
  - Exact run-time depends on the run-time of their parameter functions

More on constants and local variables

- When you assign a value to a variable inside a function, that variable is local to that function.
- You can define constants outside a function, but you cannot change them inside the function.

```
# Variables declared outside fn - can we use them in fn?
tax_rate = 0.15
greeting = "hi"
my_rate = tax_rate * 2

# fn_one: None -> None
def fn_one():
    # We can use the values declared outside
    my_rate = tax_rate / 2
    # Note that my_rate is now local to fn
    # We can no longer use the other value of my_rate
    # inside fn_one

    print greeting  ## (*)
    # The following causes an error at (*)
    # because greeting is now a local variable
    # instead of a global constant
    #greeting = "Aloha"
```

More on parameters

- If a parameter receives a new value inside a function, that change is local only.
- If a parameter is a list, any changes made to the list contents are still in effect when the function is completed.

```
# fn_two: (listof Y) (listof Z) X -> None
def fn_two(L,M,x):
    x = 10
    L = "Howdy"

    M[0] = 'abc'
    M.append(x)

# Call the function
A = []
B = [1,2,3]
z = 42.42
fn_two(A,B,z)
print A, B, z
```

Memory Model Principles

1. Memory model:
   - does a variable hold an atomic value or a pointer to a complex value?
2. A parameter always gets a copy of the value of the expression passed as an argument.
   - If this expression is a pointer, the parameter will point to the same complex object.
3. Creating a new complex object or atomic variable is local.

Goals of Module 6

- We should now be able to write any of our Scheme programs in Python, using
  - Strings and their methods
  - Lists and their methods
  - Lists used to implement structures
  - Mutation of lists
  - Functional abstraction and **`lambda`**