Module 7: Strings, lists and abstract list functions

Topics:

- Strings and their methods
- Lists and their uses
- Mutating lists
- Abstract list functions

Readings: ThinkP 8, 10

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" => 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')
```

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"`**

Considering **`userid`** again

What if userid accepted a single string, such as "Harry James Potter"?

```
name.split()
>>> ["Harry", "James", "Potter"]
```

This is a list of strings – how can we use this?

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]]`

Another useful list function

- **`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!

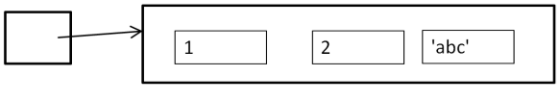
Functions vs Methods

- 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

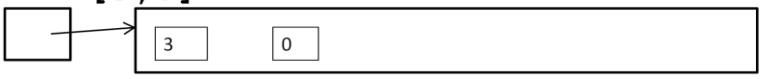
```
L = [1,2,'abc']
```



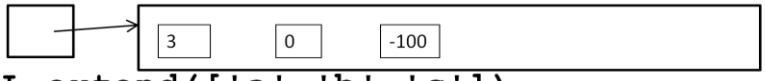

```
L[1] = 3
L[0] = -L[1]
L[2] = True
```

Other ways to mutate a list

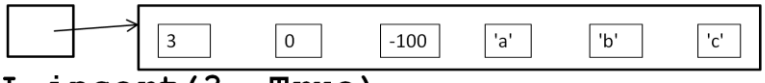
```
L = [3,0]
```



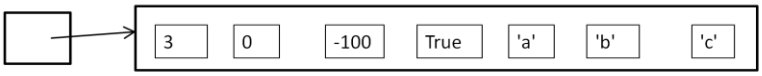

```
L.append(-100)
```




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




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



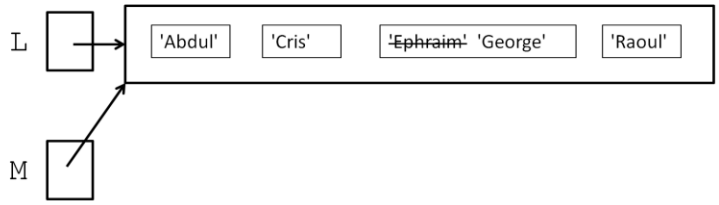
Aliasing and Lists

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

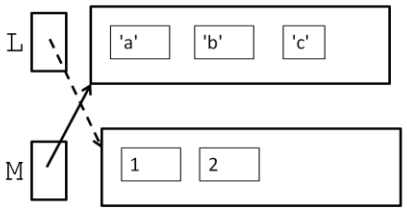
```
L=['Abdul','Cris','Ephraim','Raoul']
M = L
M[2] = 'George'
```



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,
  - Modify an existing list, or
  - Create and return a new list
- Review ThinkP 10.12

More on constants and local variables

- When you assign a value to a variable inside a function, that variable to 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

    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
```


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.

Mutable and Immutable Values in Python

- Numbers are immutable
- Strings are immutable
- Lists are mutable
- List-like objects called tuples are immutable
- Most other kinds of complicated data storage are mutable

Testing Mutation

1. Set values of state variables
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 state variable with its expected value

Example: Mutation

```
import check
def drop_x_y(reduction):
     global x,y
     x = x - reduction
     y = x - reduction
## Q4, Test 1: drop x from 5->3, y from 10->1
x = 5
y = 10
check.expect("Q4T1", drop_x_y(2), None)
check.expect("Q4T1(x)", x, 3)
check.expect("Q4T1(y)", y, 1)
```

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)
```


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 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

Goals of Module 7

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