CISC 108: Introduction to Computer Science I

# Python, Part 2

Department of Computer and Information Sciences
University of Delaware

December 1, 2016

# Lists

- Python has several "sequence" data structures similar to ISL lists
- `list`
  - `[item0, item1, item2]`
  - a mutable sequence of data
  - (we will see what "mutable" means soon)
- `tuple`
  - `(item0, item1, item2)`
  - an immutable sequence of data
  - note: singleton tuple must be written: `(item0,)`

We're going to use `list`s.

# Operations on lists or tuples

| | |
|---|---|
| `x in s` | True if an item of s is equal to x, else False |
| `x not in s` | False if an item of s is equal to x, else True |
| `s + t` | the concatenation of s and t |
| `s * n or n * s` | equivalent to adding s to itself n times |
| `s[i]` | ith item of s, origin 0 |
| `s[i:j]` | slice of s from i to j |
| `s[i:j:k]` | slice of s from i to j with step k |
| `len(s)` | length of s |
| `min(s)` | smallest item of s |
| `max(s)` | largest item of s |
| `s.index(x[, i[, j]])` | index of the first occurrence of x in s (at or after index i and before index j) |
| `s.count(x)` | total number of occurrences of x in s |

https://docs.python.org/3/library/stdtypes.html

# Translation of ISL primitives

| ISL | Python |
|---|---|
| `(list 1 2 3)` | `[1, 2, 3]` |
| `(list 1)` | `[1]` |
| `empty` | `[]` |
| `(first alist)` | `alist[0]` |
| `(rest alist)` | `alist[1:]` |
| `(empty?  alist)` | `not alist` |
| `(cons?  alist)` | `isinstance(alist, list) and alist` |
| `(length alist)` | `len(alist)` |

# Translation of ISL list functions

```
empty = []

def cons(x, a_list):
    return [x]+a_list

def first(a_list):
    return a_list[0]

def rest(a_list):
    return a_list[1:]

def is_empty(a_list):
    return not a_list
```

# Example: Adding a number to every element of a list

```
(define (add-to-all lon n)
  (cond
    [(empty? lon) empty]
    [(cons? lon)
     (cons (+ (first lon) n) (add-to-all (rest lon) n))]))
```

```
def add_to_all(lon, n):
    if is_empty(lon):
        return empty
    else:
        return cons(first(lon) + n, add_to_all(rest(lon), n))
```

# Example: Keep even numbers in a list

```
(define (keep-evens lon)
  (cond
    [(empty? lon) empty]
    [(even? (first lon))
     (cons (first lon) (keep-evens (rest lon)))]
    [else (keep-evens (rest lon))]))
```

```
def keep_evens(lon):
    if is_empty(lon):
        return empty
    elif first(lon)%2==0:
        return cons(first(lon), keep_evens(rest(lon)))
    else:
        return keep_evens(rest(lon))
```

# Functions as data

Functions can be passed as arguments, just like any other type of data:

```
def add2(f):
    """
    Consumes:
      (num -> num) f : function on numbers
    Produces: f(1)+f(2)
    """
    return f(1)+f(2)

def times2(x):
    """
    Consumes:
      num x : any int or float
    Produces: 2*x
    """
    return 2*x

print(add2(times2))
```

yields 6.

# Local functions

Function definitions in a local scope are also possible:

```
def super_local(los):
    """
    Consumes:
      [List-of str] : list of strings
    Produces: list of strings obtained by prepending "SUPER-"
    to each element in los
    """
    def make_super(astring):
        return "SUPER-" + astring
    if is_empty(los):
        return empty
    else:
        return cons(make_super(first(los)), super_local(rest(los)))

assert super_local(["cool", "CISC108"]) == ["SUPER-cool", "SUPER-CISC108"]
```

# Lambda!

Syntax:

```
lambda var1, var2, ..., varn : expr
```

Example:

```
def add2(f):
    return f(1)+f(2)

print(add2(lambda x: 2*x))
```

# Example: define `map`

```
def my_map(f, lox):
    """
    Consumes:
      (X -> Y) f
      [List-of X] lox
    Produces: list obtained by applying f to each element of lox
    """
    if is_empty(lox):
        return empty
    else:
        return cons(f(first(lox)), my_map(f, rest(lox)))

assert my_map(lambda x: x+1, [1,2,3]) == [2,3,4]
```

# Example: define `filter`

```
def my_filter(f, lox):
    """
    Consumes:
      (X -> boolean) f
      [List-of X] lox
    Produces: sub-list of lox consisting of all those elements x for
    which f(x) is true
    """
    if is_empty(lox):
        return empty
    elif f(first(lox)):
        return cons(first(lox), my_filter(f, rest(lox)))
    else:
        return my_filter(f, rest(lox))

assert my_filter(lambda x: x%2==0, [1,2,3,4]) == [2,4]
```

# Example: define `reduce`

```
def my_reduce(f, start, lox):
    """
    Consumes:
      (X Y -> Y) f : combining function
      Y start : initial value
      [List-of X] lox
    Produces: f(...(f(f(start,x[0]),x[1]), ...), x[n])
    """
    if is_empty(lox):
        return start
    else:
        return f(first(lox), my_reduce(f, start, rest(lox)))

assert my_reduce(lambda x, y: x+y, 0, [1,2,3]) == 6
```

# List of structures!

Design function which computes total cost of list of snakes:

```
BILL  = Snake("Bill", 10, "green")
SALLY = Snake("Sally", 35, "blue")
SL2 = [BILL, SALLY]

def total_cost(snake_list):
    """
    Consumes:
      [List-of Snake] snake_list
    Produces: sum of prices of snakes
    """
    if is_empty(snake_list):
        return 0
    else:
        return first(snake_list).get_price() + \
               total_cost(rest(snake_list));

assert total_cost(SL2) == BILL.get_price() + SALLY.get_price()
```

# Better: use `reduce`

```
def total_cost_v2(snake_list):
    return my_reduce(lambda s, t: s.get_price() + t, 0, snake_list)

assert total_cost_v2(SL3) == total_cost(SL3)
```

# Do something to each element in a list

```
def what_they_say(animal_list):
    """Given list of Animals, prints what they are all saying"""
    for animal in animal_list:
        print(animal.says())

what_they_say(AL1)
```

- uses a new command: `for` ... `in`
- this *iterates* over each element of a list
- more about this next week
- note that invoking method `says()` on the animal automatically dispatches to the appropriate function
  - determined by the type of `animal`

# Total cost, revisited

```
def total_cost(alist):
    if is_empty(alist):
        return 0
    else:
        return first(alist).get_price() + total_cost(rest(alist));
```

- this function previously defined for list of Snakes
- works as is for any list of Animals
- every Animal must implement a `get_price()` method

# Filter on list of structures

```
def extract_tigers(animal_list):
    """
    Consumes:
      [List-of Animal] animal_list
    Produces: list of Tigers occurring in animal_list
    """
    return my_filter(lambda a: isinstance(a, Tiger), animal_list)

assert extract_tigers(AL1) == [TONY]
```

# Example of a `map` on list of structures

- Given list of Animals, return list of their names, but for a Dillo use "Some Dillo" in place of the name...

```
def names_of(animal_list):
    def name_of(animal):
        if isinstance(animal, Dillo):
            return "Some Dillo"
        else:
            return animal.get_name()
    return my_map(name_of, animal_list)

assert names_of(AL1) == ["Samantha", "Tony", "Some Dillo"]
```

# Lists in structures: re-do Store

```
class Store:
    """
    String name : name of store
    [List-of Animal] animal_list : stock
    """
    def __init__(self, name, animal_list):
        self.name = name
        self.animal_list = animal_list
    def get_name(self):
        return self.name
    def get_animal_list(self):
        return self.animal_list
    def stock_value(self):
        return total_cost(self.get_animal_list())
    def __str__(self):
        return self.get_name() + ":\n" + \
               my_reduce(lambda x,y: str(x) + "\n" + y, "",
                         self.animal_list)
```

# Mutation!

- a new concept: memory (state)
  - until now, all functions have been *pure functions*, like in math
  - given same input, they will always return the same result
  - the result returned is entirely independent of anythingthat has happened in the past
  - the functions have no "memory"

# Example: no state

```
;; sum : (listof number)  ->  number
;; to compute the sum of the numbers on alon0
(define (sum alon0)
  (local (;; accumulator is the sum of the numbers that preceded
  ;; those in alon on alon0
  (define (sum-a alon accumulator)
    (cond
      [(empty? alon) accumulator]
      [else (sum-a (rest alon) (+ (first alon) accumulator))])))
    (sum-a alon0 0)))
```

- even the local function with accumulator `sum-a` is still functional
- if given the same values for `alon` and `accumulator` it will always return the same result.
- THE RESULT ONLY DEPENDES ON THE INPUTS.

Sometimes, this is not enough. Examples...

# Example: state required

A simple counter:

```
>>> c = Counter(0)
>>> c.count()
    0
>>> c.count()
    1
>>> c.count()
    2
```

- same method called three times in a row
- same arguments (just implicit argument `self`)
- different result each time
- the method remembers how many times it has been called

# Address Book

```
>>> B1 = AddressBook("Personal Contacts")
>>> B1.lookup("Steve")
    'not found'
>>> B1.add("Steve", "steve@stephensiegel.org")
>>> B1.lookup("Steve")
    'steve@stephensiegel.org'
```

- two calls to method `lookup` with same arguments
- different response
- the intervening call to `add` changed the state of `B1`
- memory

# What is state?

- each scope maintains a mapping of variable names to values
- the values can change
- each object has its own scope
- there is also the global (file) scope
- there are also local scopes within each function definition
- all of these scopes have state that can be changed

# Changing state of global scope

```
# Global state changing...
X = 1
print(X)
X = X+1
print(X)
```

Output:

```
1
2
```

# Changing state of an object: Counter

```
class Counter:
    def __init__(self, initial_val):
        self.val = initial_val
    def count(self):
        result = self.val
        self.val = result + 1
        return result
```

```
>>> c = Counter(0)
>>> c.count()
    0
>>> c.count()
    1
>>> c.count()
    2
```

# Changing state of objects: multiple counters

each object has its own "memory"

```
>>> C1 = Counter(56)
>>> C2 = Counter(397)
>>> C1.count()
    56
>>> C2.count()
    397
>>> C1.count()
    57
>>> C2.count()
    398
```

# Address Book: Entry class

- An Entry represents one entry in the address book
- An entry consists of a name and an email address, both strings

```
class Entry:
    def __init__(self, name, addr):
        self.name = name
        self.addr = addr
    def get_name(self):
        return self.name
    def get_addr(self):
        return self.addr
    def __str__(self):
        return "<" + self.get_name() + ", " + self.get_addr() + ">"
```

# Address Book: main class

```
class AddressBook:
    def __init__(self, book_name):
        self.book_name = book_name
        self.entries = []
    def lookup(self, name):
        for entry in self.entries:
            if (entry.get_name() == name):
                return entry.get_addr()
        return "not found"
    def add(self, name, addr):
        self.entries.append(Entry(name, addr))
    def __str__(self):
        return "Address Book " + book_name + " with " + \
               str(len(self.entries)) + " entries"
```

```
>>> B1 = AddressBook("Personal Contacts")
>>> B1.lookup("Steve")
    'not found'
>>> B1.add("Steve", "steve@stephensiegel.org")
>>> B1.lookup("Steve")
    'steve@stephensiegel.org'
```

# Sharing: AddressBook example

```
>>> B1 = AddressBook("Personal Contacts")
>>> B1.add("Steve", "steve@stephensiegel.org")
>>> B1.add("Sam", "sam@sam.org")
>>> B1.add("Bill", "bill@bill.org")
>>> print(B1.entries)
    [<Steve, steve@stephensiegel.org>,
     <Sam, sam@sam.org>, <Bill, bill@bill.org>]
>>> B2 = AddressBook("Work Contacts")
>>> B2.entries.append(B1.entries[0])
>>> print(B2.entries)
    [<Steve, steve@stephensiegel.org>]
>>> B1.entries[0].name = "STEVERINO"
>>> print(B1.entries)
    [<STEVERINO, steve@stephensiegel.org>,
     <Sam, sam@sam.org>, <Bill, bill@bill.org>]
>>> print(B2.entries)
    [<STEVERINO, steve@stephensiegel.org>]
```

Why did the Entry in B2 change?

# References

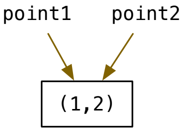
- a reference to an object can be thought of as something that "points" to that object, rather than the object itself
- in Python, operations on objects and lists happen through references
- two different variables may hold a reference to the same object
  - in the following code:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __str__(self):
        return "(" + str(self.x) + "," + str(self.y) + ")"
    def __repr__(self):
        return self.__str__()
point1 = Point(1,2)
point2 = point1
```

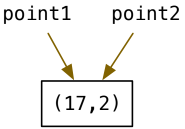
  - only one instance of `Point` is created
  - `point1` holds a reference to the new `Point` object
  - the assignment assigns this reference to `point2`
  - now both variables hold reference to the same object

## Aliased

```
point1 = Point(1,2)
point2 = point1
```

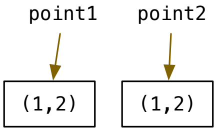



```
point1.x = 17
```

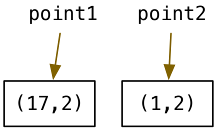



## Non-aliased

```
point1 = Point(1,2)
point2 = Point(1,2)
```




```
point1.x = 17
```



- Exercise: what happens in the following?

```
point1 = Point(1,2);
point2 = point1;
point3 = Point(3,4);
point1 = point3;
```

# Lists

Lists are also manipulated by reference

```
>>> l1 = [1,2,3,4,5]    # this creates a new list
>>> l2 = l1             # now l2 points to the same list
>>> l1
    [1, 2, 3, 4, 5]
>>> l2
    [1, 2, 3, 4, 5]
>>> l1[0]=99999
>>> l1
    [99999, 2, 3, 4, 5]
>>> l2
    [99999, 2, 3, 4, 5]
```

Moral:

- for objects and lists, a=b only assigns a a reference to b
- it does not copy the whole object b!!!

# Equality

Two notions of equality:

- `is`
  - `a is b` tells you if two references are the same
  - meaning: `a` and `b` point to the same object
  - if `a is b` then certainly `a == b`
  - but if `a == b` then maybe `a is b` or maybe not
- `==` tests if two objects are equivalent in some way:
  - two numbers are `==` if they are the same
  - two strings are `==` if they have the same sequence of characters
  - two lists are `==` if they have the same length and corresponding elements are `==`
  - `==` is a matter of interpretation
    - how you interpret data to represent information
    - if you define your own class, you need to tell Python how to decide if two objects of that class are considered `==`
    - define a method `__eq__(self, other)`

# Point class with eq method

```
class Point:
    def __init__(self,x,y):
        self.x = x
        self.y = y
    def __str__(self):
        return "("+str(self.x)+","+str(self.y)+")"
    def __repr__(self):
        return self.__str__()
    def __eq__(self, other):
        return self.x==other.x and self.y==other.y

point1 = Point(1,2)
point2 = point1
point1 is point2  # True
point1 == point2  # True
point3 = Point(1,2)
point3 is point1  # False
point3 == point1  # True
```

# References and performance

- suppose
  - class `C` has 1 `int` field
  - class `D` has 1,000,000 `int` fields
  - variables `c1` and `c2` have type `C`
  - variables `d1` and `d2` have type `D`
- how long does it take to execute the assignments . . .
  - `c1 = c2;` vs.
  - `d1 = d2;`

# References and performance

- suppose
  - class `C` has 1 `int` field
  - class `D` has 1,000,000 `int` fields
  - variables `c1` and `c2` have type `C`
  - variables `d1` and `d2` have type `D`
- how long does it take to execute the assignments ...
  - `c1 = c2;` vs.
  - `d1 = d2;`
- Answer: the same (very small) amount of time
  - the assignment simply copies a reference
  - the size of the reference is small (4 bytes) and is independent of the size of the objects being referenced

# Method invocation

- suppose
  - `C` is a class
  - `x` is a variable of type `C`
    - `x` holds a reference to an instance of `C`
  - `f` is one of the methods defined in `C`
- the expression `x.f(a1,...,an)`
  - invokes method `f`
  - on the object referred to by `x`
  - with arguments `a1,...,an`
- invocation is similar to function application in Racket
  - first, the arguments are evaluated
  - the resulting values are assigned to the formal parameters in the definition of `f`
  - the body of `f` is executed
  - invocation stops when control reaches a `return` statement or end of body

# Setters

Often provide setters and getters for each field.

```
class Point:
    def __init__(self,x,y):
        self.x = x
        self.y = y
    def get_x(self):
        return self.x
    def set_x(self, x):
        self.x = x
    def get_y(self):
        return self.y
    def set_y(self, y):
        self.y = y
    def __str__(self):
        return "("+str(self.x)+","+str(self.y)+")"
    def __repr__(self):
        return self.__str__()
    def __eq__(self, other):
        return self.x==other.x and self.y==other.y
```

# Example: set/get with two objects

```
point1 = Point(0,0);
point2 = Point(0,0);

point1.set_x(1);
point1.set_y(2);
point2.set_x(3);
point2.set_y(4);

print(point1.get_x());  // prints 1
print(point1.get_y());  // prints 2
print(point2.get_x());  // prints 3
print(point2.get_y());  // prints 4
```

# Example: aliasing

```
point1 = Point(0,0);
point2 = point1;

point1.set_x(1);
point1.set_y(2);
point2.set_x(3);
point2.set_y(4);

print(point1.get_x());  // prints 3
print(point1.get_y());  // prints 4
print(point2.get_x());  // prints 3
print(point2.get_y());  // prints 4
```

# Exercise

```
point1 = Point(1,2);
point2 = Point(3,4);

point2 = point1;
point2.set_x(5);

print(point1.get_x()); // ?
print(point1.get_y()); // ?
print(point2.get_x()); // ?
print(point2.get_y()); // ?
```

A. 1 2 3 4
B. 1 2 5 4
C. 5 2 5 4
D. 5 2 5 2
E. 5 4 5 4

# Loops!

```
for x in a_list:
    do something with x
    ...
```

- iterates over each element of the list
- first, `x` is assigned element 0 of the list
- body of `for` statement is executed
- then `x` is assigned element 1 of the list
- body of `for` statement is executed
- repeat for every element of list

In general can use any iterable object in place of `a_list`

- `range(n)`: returns an iterable for ints $0, 1, \ldots n-1$

# Exercises with loops

1. write a function to print the integers from 0 to $n - 1$
2. design a function to print every element in a list
3. print every other element in a list
4. design a function to create a copy of a list of numbers
5. add up all numbers in a list of numbers (not using reduce)
6. given a list of numbers, multiply every element by 2 (mutation!)
7. given a list of Point, add 1 to every x coodinate (mutation!)
8. given a list of Point, swap the x and y coordinate of each point (mutation!)

# Clicker 1: equality

What does the following print?

```
l1 = [1, 2, 3]
l2 = [1, 2]
l2.append(3)
print(l1 is l2)
print(l1 == l2)
```

```
A. True  True
B. True  False
C. False True
D. False False
E. ERROR
```

# Clicker 2: Sharing

What does the following print?

```
l1 = [1, 2, 3]
l2 = l1
l1[1] = 9
print(l2)
```

A. `[1, 2, 3]`
B. `[1, 9, 3]`
C. `ERROR`

# Clicker 3: Sharing

What does the following print?

```
p1 = Point(1,2)
p2 = Point(3,4)
l1 = [p1, p2]
l1[1].set_x(9)
print(l1)
print(p2)
```

```
A. [(1,2), (3,4)]     (3,4)
B. [(1,2), (3,4)]     (9,4)
C. [(1,2), (9,4)]     (3,4)
D. [(1,2), (9,4)]     (9,4)
```

# Clicker 4: Sharing

What does the following print?

```
p1 = Point(1,2)
l1 = [p1, p1]
l1[0].set_y(9)
print(l1)
print(p1)
```

```
A. [(1,9), (1,2)]    (1,2)
B. [(1,2), (1,2)]    (1,9)
C. [(1,9), (1,2)]    (1,9)
D. [(1,9), (1,9)]    (1,9)
```

# Clicker 5: Sharing

What does the following print?

```
p1 = Point(1,2)
l1 = [p1, p1]
l1[0].set_y(9)
l1[1].set_y(10)
print(l1[0] is l1[1])
print(l1[0] == l1[1])
```

```
A. True   True
B. True   False
C. False  True
D. False  False
```

# Scope

What does the following print?

```
x=1
def f():
    x=2
    print(x)
f()
print(x)
```

# Scope

What does the following print?

```
x=1
def f():
    x=2
    print(x)
f()
print(x)
```

Answer: 2 1

Why? The second x is a local variable.

Its scope is the code block in which it occurs.

In general:

- any assignment to a variable in a scope implicitly declares that variable to be a local variable in that scope
- parameters are also local

# Scope

What does the following print?

```
x=1
def f():
    print(x)
f()
print(x)
```

# Scope

What does the following print?

```
x=1
def f():
    print(x)
f()
print(x)
```

Answer: 1 1

Why?

Because there is no assignment to `x` in the local block, `x` is not declared to be local. Instead `x` refers to the global `x`.

# Scope

What does the following print?

```
x=1
def f():
    y = x
    print(y)
    y = 9
    print(x)
f()
print(x)
```

# Scope

What does the following print?

```
x=1
def f():
    y = x
    print(y)
    y = 9
    print(x)
f()
print(x)
```

- 1 1 1
- y is local
- x is global
- setting y to 9 does not change what is stored in x

# The `global` keyword

What is you want to assign to the global `x`?

```
x=1
def f():
    global x
    x=2
f()
print(x)
```

- output: 2
- a statement of the form `global x` declares that references to `x` in the block will refer to the global `x`
- `global` does not carry over to sub-blocks

# The `global` keyword

```
x=1
def f():
    global x
    def g():
        x=999   # this is a local x
    g()
    print(x)    # this is the global x
f()
```

- output: 1

# The `global` keyword

```
x=1
def f():
    global x
    def g():
        global x
        x=999   # this is the global x
    g()
    print(x)    # this is also the global x
f()
```

- output: 999

# The `nonlocal` keyword

To refer to a local variable in a super-scope, use `nonlocal`:

```
x=1 # the global x
def f():
    x=2 # a local x
    def g():
        nonlocal x
        x=3  # the local x in scope above
    g()
    print(x)  # prints 3
f()
print(x)  # still 1
```

- if you write `nonlocal x` there must be a local `x` in one of your super-scopes (else: ERROR)
- all uses of `x` in your scope will refer to the local `x` in the smallest super-scope

# `while` loops

- sometimes you want to loop but you don't know how many times you want to loop ahead of time
- you want to loop as long as some condition holds
- `for .. in` is not appropriate in this case
- instead, use `while`

```
def mylog(n):
    count = 0
    while n>1:
        n = n/2
        count = count + 1
    return count
```

- checks the condition `n>1`
- if condition is False, exit the loop
- if condition is True, execute the loop body, repeat
- careful: it is possible to loop forever!

# why `while` loops are the only loops you need

```
lon = [1,2,3]
```

```
for i in range(len(lon)):
    lon[i] = 2*lon[i]
```

and

```
i=0
while i<len(lon):
    lon[i] = 2*lon[i]
    i = i+1
```

do the exact same thing.

# Estimate $\pi$ to within some tolerance

Exercise: write function to approximate $\pi$ to within a given tolerance using the formula.

$$\pi = 4 \sum_{i=1}^{\infty} \frac{(-1)^{i+1}}{2i-1}$$
$$= 4(1/1 - 1/3 + 1/5 - 1/7 + \ldots).$$

When two successive partial sum are within tolerance, stop.
You can either `return` value immediate or `break` out of loop.

# Checking input

The `raise` statement:

- If something goes wrong, `raise` an exception.
- There are various kinds of exceptions.
- We will use `ValueError`:
- `raise ValueError("Some nice error message")`
- Typical pattern: check inputs satsify some condition.
- If they don't, raise a `ValueError`.