I. Lambda functions

a. Python supports the creation of anonymous functions (i.e. functions that are not bound to a name) at runtime, using a construct called "lambda".

```
>>> def f (x): return x**2
...
>>> print f(8)
64
>>>
>>> g = lambda x: x**2
>>>
>>> print g(8)
64
```

As you can see, f() and g() do exactly the same and can be used in the same ways. Note that the lambda definition does not include a "return" statement -- it always contains an expression which is returned.

b. The below code defines a function "`make_inrementor`" that creates an anonymous function on the fly and returns it. The returned function increments its argument by the value that was specified when it was created.

```
>>> def make_incrementor (n): return lambda x: x + n
>>>
>>> f = make_incrementor(2)
>>> g = make_incrementor(6)
>>>
>>> print f(42), g(42)
44 48
```

II. Environment Diagrams

a. An environment in which an expression is evaluated consists of a sequence of *frames*, depicted as boxes. Each frame contains *bindings*, each of which associates a name with its corresponding value. There is a single *global* frame. Assignment and import statements add entries to the first frame of the current environment.

b. Online Python Tutor: http://pythontutor.com/visualize.html

c. Lets draw the environment diagram for the following;

```
from operator import add
def square(x):
    """Return x squared."""
    return x * x
square(2)
```

III. Sequence

a. A sequence is an ordered collection of data values. Unlike a pair, which has exactly two elements, a sequence can have an arbitrary (but finite) number of ordered elements.

b. Two properties of sequences

   i. **Length** - A sequence has a finite length.

   ii. **Element selection** - A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

c. Tuples

i. The `tuple` is itself a full sequence type, which can be constructed by separating values by commas. Although not strictly required, parentheses almost always surround tuples.

```
>>> (1, 2)
(1, 2)
>>> pair = (1, 2)
>>> pair
(1, 2)
>>> pair[0]
1
```

ii. Tuples can have arbitrary length, and they exhibit the two principal behaviors of the sequence abstraction: length and element selection.

iii. `digits` is a tuple with four elements.

```
>>> digits = (1, 8, 2, 8)
>>> len(digits)
4
>>> digits[3]
8
```

iv. tuples can be added together and multiplied by integers. For tuples, addition and multiplication do not add or multiply elements, but instead combine and replicate the tuples themselves.

```
>>> (2, 7) + digits * 2
(2, 7, 1, 8, 2, 8, 1, 8, 2, 8)
```

IV. Immutable objects
- a. Values never change
- b. Examples: numbers, Booleans, tuples, ranges, and strings

V. Mutable Objects
- a. Can change throughout the execution of a program

VI. Mutable Objects: Lists
- a. Method calls and assignment statements can change the contents of a list.
- b. The Python language does not give us access to the implementation of lists, only to the sequence abstraction and the mutation methods we have introduced in this section. To overcome this language-enforced abstraction barrier, we can develop a functional implementation of lists, **rlists** (recursive lists).
- c. Lets draw the environment diagram that illustrates the structure of the recursive representation of a four-element list: `1`,`2`, `3`, `4`.

```
>>> up_to_four = (1, (2, (3, (4, None))))
```

- d. This nested structure corresponds to a very useful way of thinking about sequences in general. A non-empty sequence can be decomposed into:
  - i. its first element, and
  - ii. the rest of the sequence.
- e. Since our list representation is recursive, we will call it an `rlist` in our implementation, so as not to confuse it with the built-in `list` type in Python. The value `None` represents an empty recursive list.

```python
empty_rlist = None

def rlist(first, rest):
    """Construct a recursive list from its first element and the
    rest."""
    return (first, rest)

def first(s):
    """Return the first element of a recursive list s."""
    return s[0]

def rest(s):
    """Return the rest of the elements of a recursive list s."""
    return s[1]

def len_rlist(s):
    if s == empty_rlist:
        return 0
    return 1 + len_rlist(rest(s))

def getitem_rlist(s, k):
    if k == 0:
        return first(s)
    return getitem_rlist(rest(s), k - 1)
```

f. These two selectors, one constructor, and one constant together implement the recursive list abstract data type.

g. We can use the constructor and selectors to manipulate recursive lists.

```
>>> counts = rlist(1, rlist(2, rlist(3,
rlist(4, empty_rlist))))
>>> first(counts)
1
>>> rest(counts)
(2, (3, (4, None)))
```

VII. Mutable Objects: Dictionaries

a. Dictionaries are Python's built-in data type for storing and manipulating correspondence relationships. A dictionary contains key-value pairs, where both the keys and values are objects. The purpose of a dictionary is to provide an abstraction for storing and retrieving values that are indexed not by consecutive integers, but by descriptive keys.

b. Strings commonly used as keys. This dictionary literal gives the values of various Roman numerals.

```
>>> numerals = {'I': 1.0, 'V': 5, 'X': 10}
```

c. Looking up values by their keys uses the element selection operator that we previously applied to sequences.

```
>>> numerals['X']
10
```

d. A dictionary can have at most one value for each key. Adding new key-value pairs and changing the existing value for a key can both be achieved with assignment statements.

```
>>> numerals['I'] = 1
```

```
>>> numerals['L'] = 50
>>> numerals
{'I': 1, 'X': 10, 'L': 50, 'V': 5}
```

e. Restrictions of Dictionaries
    i. A key of a dictionary cannot be an object of a mutable built-in type.
    ii. There can be at most one value for a given key.

VIII. Objects and Classes

a. Object-oriented programming (OOP) is a method for organizing programs.
b. Like abstract data types, objects create an abstraction barrier between the use and implementation of data.
c. A class serves as a template for all objects whose type is that class. Every object is an instance of some particular class.
d. A class definition specifies the attributes and methods shared among objects of that class. We will introduce the class statement by visiting the example of a bank account.
e. What methods (actions) does a bank account need? Withdraw, deposit, etc.
f. An `Account` class allows us to create multiple instances of bank accounts. The act of creating a new object instance is known as *instantiating* the class.
g. User-defined classes are created by `class` statements,

```
class <name>(<base class>):
    <suite>
```

h. When a class statement is executed, a new class is created and bound to `<name>` in the first frame of the current environment.
i. The `<suite>` of a `class` statement contains `def` statements that define new methods for objects of that class. The method that initializes objects has a special name in Python, `__init__` (two underscores on each side of "init"), and is called the *constructor* for the class.

```
>>> class Account(object):
        def __init__(self, account_holder):
            self.balance = 0
            self.holder = account_holder
```

j. The `__init__` method for `Account` has two formal parameters. The first one, `self`, is bound to the newly created `Account`object. The second parameter, `account_holder`, is bound to the argument passed to the class when it is called to be instantiated.
k. The syntax in Python for instantiating a class is identical to the syntax of calling a function. In this case, we call `Account` with the argument `'Jim'`, the account holder's name.

```
>>> a = Account('Jim')
```

l. Now, we can access the object's `balance` and `holder` using dot notation.

```
>>> a.balance
0
>>> a.holder
'Jim'
```

m. Each new account instance has its own balance attribute, the value of which is independent of other objects of the same class

```
>>> b = Account('Jack')
>>> b.balance = 200
>>> [acc.balance for acc in (a, b)]
[0, 200]
```