Using Python’s Data Structures
And Making Your Own

(Introduction to Classes and Objects)

Lists

```python
>>> m = [23, 42, 89]
>>> print m
[23, 42, 89]
>>> m[0] = 28
>>> print m
[28, 42, 89]
>>> m.append(132)
>>> print m
[28, 42, 89, 132]
>>> 
```

Access individual elements by index

Add to the end

List values

The value of m is a reference to the list containing 23, 46, 15, 12

The stack (of activation records)

The heap (of data structures)

Copying references

m = [23, 46, 15, 12]

n = m
Changing shared values

\[ m = [23, 46, 15, 12] \]
\[ n = m \]
\[ n[2] = 88 \]

Activation records: Variables and values

```python
def foo(x):
    y = x + 1
    return y
def bar():
    y = 42
    z = foo(y)
```

Passing values

```python
def change(x, m):
    x = 3
    m[1] = 88
def bar():
    y = 42
    k = [99, 97]
    change(y, k)
```

```python
def change(x, m):
    x = 3
    m[1] = 88
def bar():
    y = 42
    k = [99, 97]
    change(y, k)
```
def change(x, m):
    x = 3
    m[1] = 88

def bar():
    y = 42
    k = [99, 97]
    change(y, k)

y
k
42
99
97
bar
99
88

Passing lists to functions

>>> def changeMyVariables(x, m):
...     x = 32
...     m.append("gotcha")
...     ...
...     y = 48
...     n = ["no", "way"]
...     changeMyVariables(y, n)
...     print y, n
48 ['no', 'way', 'gotcha']
>>> print y
48
>>> print n
['no', 'way', 'gotcha']

What are those references, really?

References are addresses (locations),
used to select cells in the memory chips
Funny notation for a function

```python
m = [24, 38]
m.append(99)
print m
```

append is really a function with 2 arguments, `m` and 99
A “method” of the list object

Classes: Do-it-yourself types

```python
def MyNewClass:
    """Just because""
    def __init__(self):
        """Here’s how I create a new object""
        self.a_field = 42
    def a_method(self, arg):
        """Do something to the object""
        self.field = self.field + arg

my_obj = MyNewClass()  # Create one
my_obj.a_method(99)    # Do something to it
```

```python
class TicTacBoard:
    """A 3x3 board for tic-tac-toe, also known as naughts and crosses."
    def __init__(self):
        """Constructor creates a TicTacBoard object"
        self.elements = [
            [' ', ' ', ' '],
            [' ', ' ', ' '],
            [' ', ' ', ' ']]
# Rows and columns are numbered 1, 2, 3
def row(self, i):
    """Return the i'th row (1, 2, or 3) of the board as a list"
    assert i >= 1 and i <= 3
    return self.elements[i-1]
```
class TicTacBoard:
...

def move(self, row, col, mark):
    """Place mark at (row, col)"""
    assert row >= 1 and row <= 3
    assert col >= 1 and col <= 3
    rowcols = self.elements[ row - 1 ]
    rowcols[ col - 1 ] = mark

    board.move( 1, 3, "X" )

---

Summary

Python “container” types are references
Assignment copies the reference (arrow), not the contents of the container
Two variables can reference (“point to”) the same object. Changes to one affect the other.
We can define our own container objects by defining classes
Objects contain (references to) their methods as well as data
m.foo(x) passes m as the “self” reference