CSc 120
Introduction to Computer Programming II

10: References
• Consider the following code. What does it do? How long does the highlighted line take to run?

```python
shortStr = "short"
longStr = "this is a longer string"
l = ["foo", shortStr, "bar", "baz"]
l[1] = longStr
```
How Long Does This Take?

shortStr = “short”
longStr = “this is a longer string”
l = [“foo”, shortStr, “bar”, “baz”]
l[1] = longStr

How do we fit a longer string into a slot that used to hold a short one???

“foo” “short” “bar” “baz”

“foo” “this...” “bar” “baz”
Consider the following snippet. Is either object “inside” the other object?

class List:
    __init__(self, val):
        self.next = None
        self.prev = None
        self.val = val

first = List(10)
second = List("word")
first.next = second
second.prev = first
What In the World???

class List:
    __init__(self, val):
        self.next = None
        self.prev = None
        self.val = val

first = List(10)
second = List("word")
first.next = second
second.prev = first
class List:
    __init__(self, val):
        self.next = None
        self.prev = None
        self.val = val

first = List(10)
second = List("word")
first.next = second
second.prev = first

More precise: each next/prev reference points at the beginning of the other object.
References

• A **reference** is a variable which holds the **location** of another variable in memory.
  - It does **NOT** contain the variable itself!

• In Python, **all variables** are references to objects (even `None`, integers, etc.)
  - Also, all elements of a list
  - Also, all fields in a class
  - Also, all values in a tuple, dictionary, or set

This is unusual. Most programming languages only use references for some variables.
I'm amazed that even these are references...

\[
x = 1 \\
y = 2 \\
x = x + y
\]
I'm amazed that even these are references...

\[
\begin{align*}
  x &= 1 \\
  y &= 2 \\
  x &= x + y
\end{align*}
\]
I'm amazed that even these are references...

\[ x = 1 \]
\[ y = 2 \]
\[ x = x + y \]
I'm amazed that even these are references...

\[
\begin{align*}
x &= 1 \\
y &= 2 \\
x &= x + y
\end{align*}
\]

Python automatically cleans up objects that are no longer in use. This is known as garbage collection, and it's completely automatic.
shortStr = "short"
longStr = "this is a longer string"
l = ["foo", shortStr, "bar","baz"]
l[1] = longStr

- **Aliasing** is when multiple references point to the same object.
- This happens twice in the example above.
shortStr = "short"
longStr = "this is a longer string"
l = ["foo", shortStr, "bar", "baz"]
l[1] = longStr
shortStr = "short"
longStr = "this is a longer string"
l = ["foo", shortStr, "bar", "baz"]
l[1] = longStr
shortStr = "short"
longStr = "this is a longer string"
l = ["foo", shortStr, "bar", "baz"]
l[1] = longStr

shortStr

longStr

l
shortStr = "short"
longStr = "this is a longer string"
l = ["foo", shortStr, "bar", "baz"]
l[1] = longStr

shortStr

longStr

"short"
"bar"
"foo"
"baz"
"this is a longer string"
Changing a Reference

We can replace the short string with the long **cheaply** because all we do is to **update the reference**.

```
shortStr
  ↓
  1
longStr
```

```
“short”
“bar”
“foo”
“baz”
“this is a longer string”
```
Changing a Reference

- All references are the same size
  - 8 bytes (64 bits) is common on modern computers
- Changing a reference is $O(1)$
  - Compare this to inserting into a list: $O(n)$
    - Changing a reference changes one slot
    - Inserting changes many
  - This is why we store strings by reference, not value!
Aliasing: Why Do We Care?

- What is the content of each list at the end of this code?

```python
list1 = [2, 3, 5]
list2 = [7, 11, 13]
list3 = list1
list4 = list1 + list2

list1[2] = -3
list3.append("foo")
list4[0] = None
```
list1 = [2, 3, 5]
list2 = [7, 11, 13]
list3 = list1
list4 = list1+list2

list1[2] = -3
list3.append("foo")
list4[0] = None

Remember, a list doesn't actually contain the values – it contains references to other objects.

But for simplicity, we often draw the object inside the list itself.
list1 = [2, 3, 5]
**list2 = [7, 11, 13]**
list3 = list1
list4 = list1+list2

list1[2] = -3
list3.append("foo")
list4[0] = None
list1 = [2, 3, 5]
list2 = [7, 11, 13]
list3 = list1
list4 = list1 + list2

list1[2] = -3
list3.append("foo")
list4[0] = None

Assigning a list to a new variable creates an alias.
list1 = [2, 3, 5]
list2 = [7, 11, 13]
list3 = list1
list4 = list1+list2

list1[2] = -3
list3.append("foo")
list4[0] = None

List concatenation creates a brand-new list. It is a duplicate.
list1 = [2, 3, 5]
list2 = [7, 11, 13]
list3 = list1
list4 = list1 + list2

list1[2] = -3
list3.append("foo")
list4[0] = None

Changing a value inside a list changes one element.
All of the aliases of the list see the same change.
list1 = [2, 3, 5]
list2 = [7, 11, 13]
list3 = list1
list4 = list1 + list2

list1[2] = -3
list3.append(“foo”)  
list4[0] = None

In the same way, .append() modifies the list (and thus all aliases of it). It does NOT create a new list!
So .append() and + are different!
list1 = [2, 3, 5]
list2 = [7, 11, 13]
list3 = list1
list4 = list1 + list2

list1[2] = -3
list3.append(“foo”)  
list4[0] = None

Yes, it's legal for one of the fields in a list to be None. The element exists – but it points at “nothing.”

In fact, in later classes, this will be quite common!
Aliasing: Why Do We Care?

- When we create an alias, there are **two** (or more) references to the same object
- Modifying the object modifies what **all** references see

- Sometimes this is desirable
- Sometimes it's a bug
Immutability

- Because aliasing can lead to bugs, Python provides some \textbf{immutable} data types
  - string
  - tuple
  - frozenset
- Immutable types can never change, so it's safe to share (efficiency)
- But you have to duplicate the object to make changes (waste)
Non-Transitive Immutability

• Caveat: Immutability is non-transitive

• A list inside a tuple can be modified
  – The reference from the tuple to the list never changes
  – But the list itself can

• This is true in many languages (but not all)
>>> x = ([],[],[])  

>>> x[0] = [1,2,3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment

>>> x[0].append(1)
>>> x[0].append(2)
>>> x[0].append(3)

>>> x
([1, 2, 3], [], [])
>>> x = ([],[],[])

>>> x[0] = [1,2,3]
Traceback (most recent call last):
  File "<stdin>" , line 1 , in <module>
TypeError: 'tuple' object does not support item assignment

>>> x[0].append(1)
>>> x[0].append(2)
>>> x[0].append(3)
>>> x
([1, 2, 3], [], [])
>>> x = ([], [], [])

>>> x[0] = [1, 2, 3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment

>>> x[0].append(1)
>>> x[0].append(2)
>>> x[0].append(3)

>>> x
([1, 2, 3], [], [])
>>> x = ([], [], [])

>>> x[0] = [1, 2, 3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment

>>> x[0].append(1)
>>> x[0].append(2)
>>> x[0].append(3)

>>> x
([1, 2, 3], [], [])
>>> x = ([], [], [])

>>> x[0] = [1,2,3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment

>>> x[0].append(1)
>>> x[0].append(2)
>>> x[0].append(3)
>>> x
([1, 2, 3], [], [])
Cycles

- **Cyclic** data structures are common in programming
  - Would be impossible if one object was “inside” another
  - Easy with references

```python
class List:
    __init__(self, val):
        self.next = None
        self.prev = None
        self.val = val

first = List(10)
second = List(“word”)
first.next = second
second.prev = first
```
Cycles

class List:
    __init__(self, val):
        self.next = None
        self.prev = None
        self.val = val

first = List(10)
second = List("word")
first.next = second
second.prev = first

Technically, None is an object – and all references to None point to that object.

But we'll ignore that in our pictures.
Cycles

class List:
    __init__(self, val):
        self.next = None
        self.prev = None
        self.val = val

first = List(10)
second = List("word")
first.next = second
second.prev = first
Cycles

class List:
    __init__(self, val):
        self.next = None
        self.prev = None
        self.val  = val

first  = List(10)
second = List(“word”)
first .next = second
second.prev = first
Cycles

```python
class List:
    __init__(self, val):
        self.next = None
        self.prev = None
        self.val  = val

first  = List(10)
second = List("word")
first .next = second
second.prev = first
```

```
first  = L
second  = L
```
Summary

- A **reference** is a variable which contains the address of another variable (instead of the value)
  - In Python, **every variable** is a reference
- All references have the same size; can be changed in $O(1)$
- An **alias** is when two or more references point to the same object
  - Change the object through any reference, and all alias see the same change!