References
Homework notes

• Homework 2 grades posted on canvas

• Homework 3 due tomorrow

• Homework 4 posted on canvas
  – Due Tuesday, Oct. 3
Style notes

• Comment your code!
  – A short line of comments per logical block of code is sufficient

• Add whitespace
  – Makes it a lot easier to parse your code, especially if it is logically separated

• Use meaningful variable names
  – If you use a/b, there’s no way to easily find out what they do
  – EVEN YOU WON’T REMEMBER IN A FEW DAYS!
Simple values vs. collections

- >>> a = 5
  >>> b = a
  >>> a = a + 1
  >>> b
  5
- >>> a = [1, 2, 3]
  >>> b = a
  >>> a[1] = 5
  >>> b
  [1, 5, 3]
- What’s going on?
Think of a variable as a “box” that can hold a small amount of information.

In the case of a simple value (number or boolean), the value can fit in the box.

In the case of a larger value, such as a list, what is put in the box is a reference to the value.

What we copy to another variable is what is in the box.
It’s all about memory

• Variables always hold small values – integers, floats, booleans, and references
  – Small values are cheap to copy and pass around
  – They are kept in a special part of memory called the stack

• All other kinds of values are larger (some of which can grow and shrink, like lists)
  – These larger values are kept in a special part of memory called the heap
  – Larger values are expensive (in both time and memory) to copy, so it doesn’t happen automatically
Heap vs. Stack

• Conceptually, two separate contiguous blocks of memory
  – May not be contiguous on an actual machine

• Variables are copied on the stack directly
  – a = 5

• More complex types contain a pointer on the stack and actual data on the heap
  – my_list = [1, 2]
Call by value

• When we call a function, the arguments in the call are evaluated, and the values are put into the parameters
  – If the parameters are changed within the function (which isn’t good style), the changes are not put back into the arguments
  – Besides, the arguments are not necessarily simple variables; they could be literal numbers, or expressions

• We refer to this by call by value

• IDLE example: callByValue.py
Call by value revisited

• \( a = 5 \)

```python
def alter(x):
    x = x + 1

def main():
    a = 5
    alter(a)
    print(a)

main()
```

Stack

```
5
```

Heap
Call by value revisited

- \(a = 5\)
- \(\text{alter}(a)\)

```python
def alter(x):
    x = x + 1

def main():
    a = 5
    alter(a)
    print(a)

main()
```

The stack diagram shows:

- Initial state:
  - \(a = 5\)

The heap diagram shows:
- Initial state:
  - \(5\)
Call by value revisited

- \( a = 5 \)
- \( \text{alter}(a) \)
- \( x = x + 1 \)

```python
def alter(x):
    x = x + 1

def main():
    a = 5
    alter(a)
    print(a)
main()
```

Stack

| 6 |

Heap
Call by value revisited

- $a = 5$
- $\text{alter}(a)$
- $x = x + 1$
- $\text{print}(a)$

```python
def alter(x):
    x = x + 1

def main():
    a = 5
    alter(a)
    print(a)

main()
```
Call by reference

• When a function is called with something other than a number or boolean, the same rules apply, but something different happens
  – What is in the argument is a reference, and that is what is passed to the function
  – The argument continues to refer to the same list but the list has been altered
• This is call by reference
• IDLE example: callByReference.py
Call by reference revisited

- \( b = [1, 2, 3] \)

```python
def alter(x):
    x[1] = 99  # changes contents of \( b \)
    x = [4, 5, 6]  # does not affect \( b \)

def main():
    b = [1, 2, 3]
    alter(b)
    print(b)
main()
```
Call by reference revisited

- \( b = [1, 2, 3] \)
- \( \text{alter}(b) \)

```python
def alter(x):
    x[1] = 99 # changes contents of b
    x = [4, 5, 6] # does not affect b

def main():
    b = [1, 2, 3]
    alter(b)
    print(b)

main()
```
Call by reference revisited

- b = [1, 2, 3]
- alter(b)
- x[1] = 99

def alter(x):
    x[1] = 99  # changes contents of b
    x = [4, 5, 6]  # does not affect b

def main():
    b = [1, 2, 3]
    alter(b)
    print(b)

main()
Call by reference revisited

- \( b = [1, 2, 3] \)
- `alter(b)`
- \( x[1] = 99 \)
- \( x = [4, 5, 6] \)

```python
def alter(x):
    x[1] = 99  # changes contents of \( b \)
    x = [4, 5, 6]  # does not affect \( b \)

def main():
    b = [1, 2, 3]
    alter(b)
    print(b)

main()
```
Call by reference revisited

- b = [1, 2, 3]
- alter(b)
- x[1] = 99
- x = [4, 5, 6]
- print(b)

def alter(x):
    x[1] = 99 # changes contents of b
    x = [4, 5, 6] # does not affect b

def main():
    b = [1, 2, 3]
    alter(b)
    print(b)

main()
Collections as parameters

• A variable is a “box” that can hold a small amount of information
  – Four bytes of actual data, but there is associated information to describe the type of the data

• Dictionaries and sets are mutable: You can change the values in them
  – Dictionaries and sets are not small, so like lists, a variable whose value appears to be a dictionary or set, is actually a reference to that dictionary or set
  – Python keeps this straight, so you don’t (usually) have to

• **Bottom line:** Dictionaries and sets are like lists; when you pass one into a function,
  – You can change the values in the dictionary or set, but
  – You can’t change it to be a different dictionary or set
Strings as parameters

• Are strings passed to functions by value or by reference?
• Answer: Strings are passed by reference, but since they are immutable, nothing you do in the function will change the original string
• IDLE example: stringParam.py
Operations on lists: making a list

• You can enter a list directly:
  • >>> [1, 2, 3]
    [1, 2, 3]
• You can give a sequence to the list function:
  • >>> list(range(1, 4))
    [1, 2, 3]
  • >>> list({'one', 'two', 'three'})
    ['two', 'three', 'one']
  • >>> list({'one': 1, 'two':2})
    ['two', 'one']
• You can “multiply” a list:
  • >>> ['A'] * 3
    ['A', 'A', 'A']
  • >>> ['a', 'b', 'c'] * 2
    ['a', 'b', 'c', 'a', 'b', 'c']
List multiplication gone wrongs

- >>> m = [['a', 'b', 'c']] * 3
  >>> m
  [['a', 'b', 'c'], ['a', 'b', 'c'], ['a', 'b', 'c']]  
  >>> m[1][2] = '*'
  >>> m
  [['a', 'b', '*'], ['a', 'b', '*'], ['a', 'b', '*']]

- Explanation:
  1. In the first line above, the list contains a reference to a list
  2. The *3 made copies of the reference, not copies of the list itself
  3. This gets assigned to m, which has three copies of the same reference
  4. The second assignment above changes the one referenced list
Copying collections

- Two ways to copy:
  - A **shallow copy** of an object makes a copy of an object that includes all the small values (including references) in the original object
  - A **deep copy** of an object makes a copy of that object that includes deep copies of all the referenced objects in the original object

- IDLE example. copyTypes.py
copy and deepcopy

- \( m = [1, [2, 3], 4] \)

```
# ptr to start of m
```

```
# ptr to list in m
```

```
3
2
4
```

Stack

Heap
copy and deepcopy

- m = [1, [2, 3], 4]
- m2 = copy.copy(m)
copy and deepcopy

- \( m = [1, [2, 3], 4] \)
- \( m2 = \text{copy.copy}(m) \)
- \( m3 = \text{copy.deepcopy}(m) \)
copy and deepcopy

- \( m = [1, [2, 3], 4] \)
- \( m2 = \text{copy.copy}(m) \)
- \( m3 = \text{copy.deepcopy}(m) \)
- \( m[1][1] = 99 \)

Stack

Heap

- \#ptr to start of \( m \)
- \#ptr to start of \( m2 \)
- \#ptr to start of \( m3 \)

- \#ptr to list in \( m \) 1
- \#ptr to list in \( m \) 4
- \#ptr to list in \( m \) 2
- \#ptr to list in \( m \) 3
- \#ptr to list in \( m3 \) 1
- \#ptr to list in \( m3 \) 99
copy and deepcopy

- \( m = [1, [2, 3], 4] \)
- \( m2 = \text{copy.copy}(m) \)
- \( m3 = \text{copy.deepcopy}(m) \)
- \( m[1][1] = 99 \)
- \( m.\text{append}(5) \)