Intro to Computer Science

Previous

- Python plumbing
  - The `print` statement
  - Comments
  - Code style
- Sequences
  - Strings
  - Lists

Readings	
Gaddis	• Chapter 7.1—7.3, 7.5 • Chapter 8

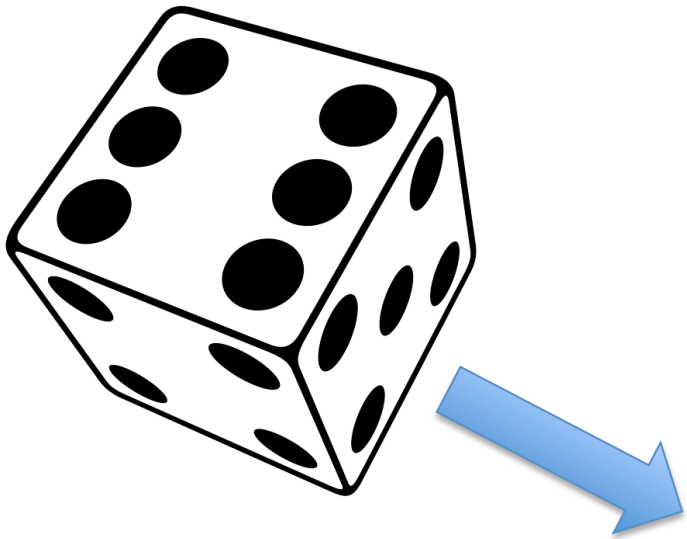
Next

- Dice game review
- More lists
  - Aliasing
  - Lists of lists
- for-loops

Readings	
Gaddis	• Chapter 4.3—4.4, 4.6 • Chapter 7 (remaining)

The beginnings of data structures

- Came up with abstract representation of physical object
  - Mapped the physical into the digital!



```
die = [ 'one', 'two', …, 'five', 'six']
```

Aliasing

- The assignment operator actually makes an *alias*
  - (Another) variable that points to a given *value*
  - Instead of "taking-on" the value, it's actually "point-to" the value
- Our lack of distinction was fine
- And then *mutability* came along…

If data is immutable, it doesn't matter how things "pointing to" it

Mutability

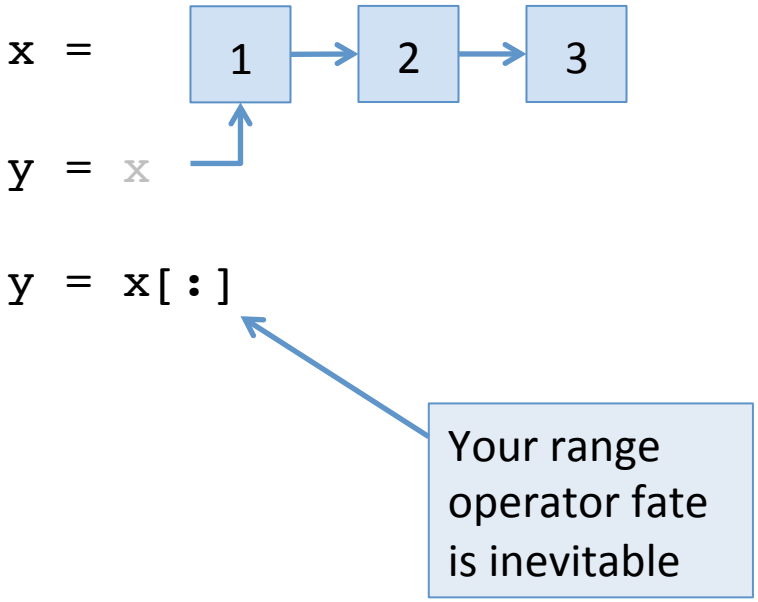
- *Mutability* refers to the ability to change a value once its bound
  - Note the use of *value* and not *variable*
  - This notion specifically refers to the contents of a variable

Python	Notes
`x = 'Hello World'` `x = 'Foodbye World'`	We can change the value of `x`
`x[0] = 'G'`	We cannot change the contents of `x` (strings are not mutable)
`x = 10` `7 = x`	This is equally strange (but illustrates the same point)
`y = [1, 2, 3]` `y[0] = 4`	Lists are mutable

Why is he telling me this?

- Because when we make an assignment to a list, we are making an alias
- The variables might have different names, but they point to the same underlying data
- So when you make a change to the underlying data…

```
x = [1, 2, 3]
y = x
x[0] = 4
print(y)
```



Iteration

- We know how to create sequences
- We know how to manipulate sequences
- How do we operate on a sequence?
  - *Iteration*: moving through a sequence, element-by-element, and performing some operation

Why is this useful?

- In class, we often perform simple examples

```
x = 1 + 2 + 3
```

- But what if we wanted to add a million numbers?

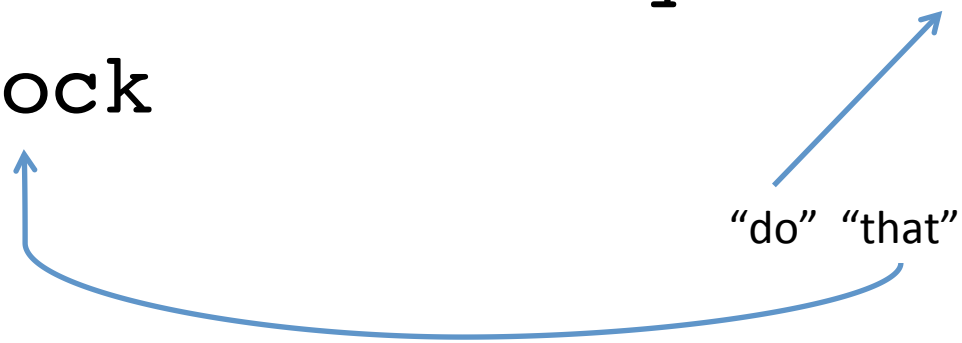
```
x = 1 + 2 + 3 + … + 1000000
```

  - You probably wouldn't sit through the class
  - And my fingers would fall off
- *Iteration!*

Iterating over a sequence

- Iterating over a sequence is so fundamental, it has its own construct in Python

```
for variable in sequence:
    block
```



- Commonly referred to as a *for-loop*

`for` syntax

`for`, `in`, and `:` are essential

"free" variable

defined sequence

```
>>>for variable in sequence:
>>>    block
```

An indentation of four spaces (most Python editors take care of this)

`for` syntax

```
>>>for variable in sequence:
>>>    block
```

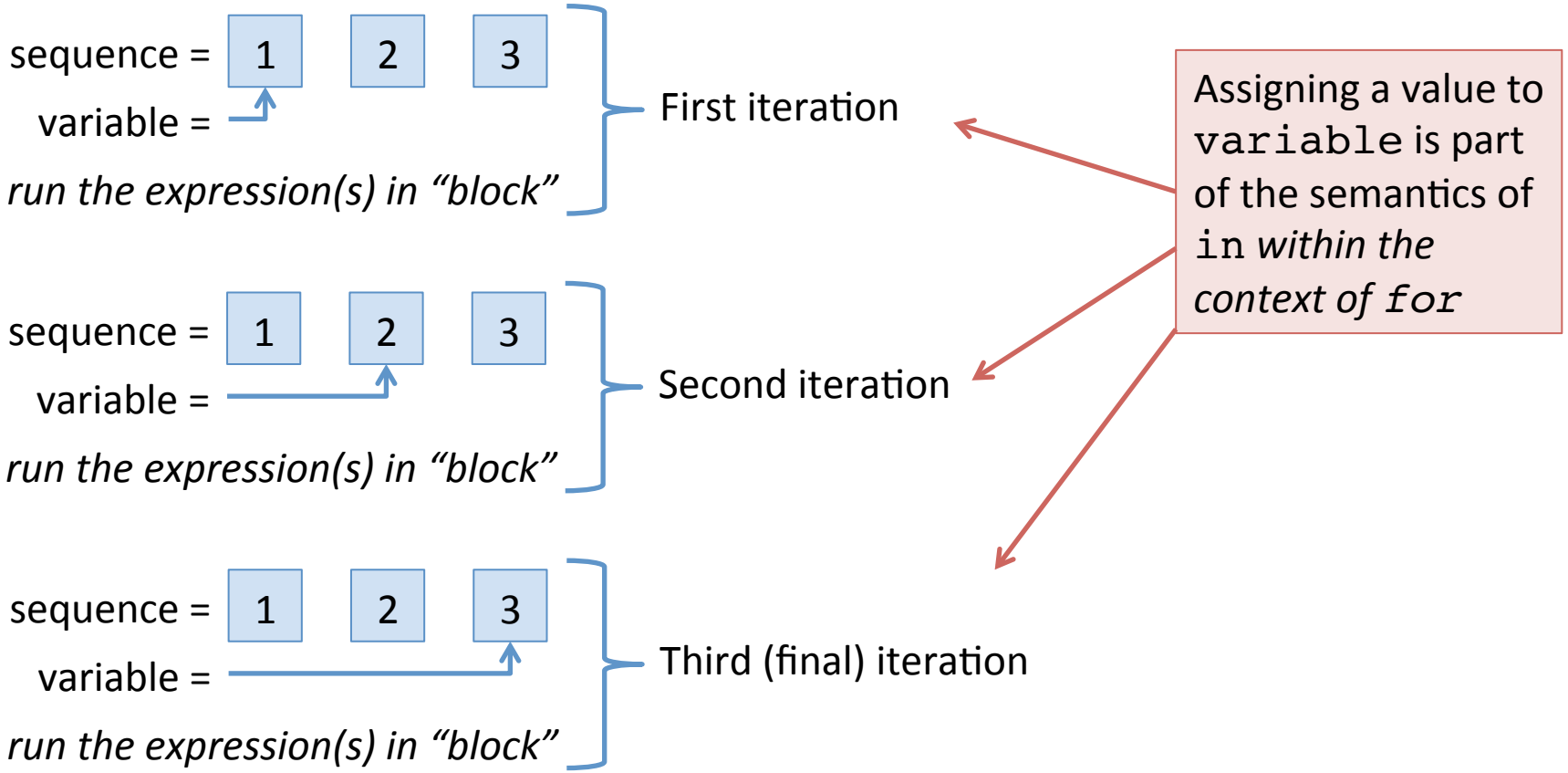
- `sequence` is a sequence that is defined

```
sequence = [1, 2, 3]
sequence = 'hello'
[1, 2, 3]
```

- Python will "move through" the sequence, one-by-one, assigning the *next* element to `variable`

`for` syntax

```
>>>for variable in sequence:
>>>    block
```



What is this block you speak of?

```
>>>for variable in sequence:
>>>    block
```

- In general, a *block* is a section of code that is grouped together
- A block can have as many lines of code as you require
  - Once Python finds the last line of the block, it goes back to the top and runs the in operator again

```
for i in [1, 2, 3]:
    print('This')
    print('is')
    print('all')
    print('the')
    print('same')
    print('block!')
    print(i)
```

block party!

```
>>>for variable in sequence:
>>>    block
```

- Python recognizes a block based on indentation
  - Generally four spaces
- Once the indentation is broken, the block is over
- In other languages, the block is demarcated with braces

  {…}

```
for i in [1, 2, 3]:
    print('My block')
    print(i)
print('done!')
```

What happens in the block…

```
>>>for variable in sequence:
>>>    block
```

- Although we move through the block several times, variables within the block retain their values
  - The only thing that gets "reset" implicitly is the free variable

```
x = 0
for i in [1, 2, 3]:
    x = x + 2
print(x)
```

Iteration	i	x
0	(undefined)	0
1	1	2
2	2	4
3	3	6

(Not so) Empty nest

```
>>>for variable in sequence:
>>>    block
```

- For-loops can contain other for-loops!
  - Known as a *nested* loop

```
for i in [1, 2, 3]:
    for j in [1, 2, 3]:
        print(i, j, i + j)
    print('done with the inner loop')
print('done with the outer loop')
```

Indentation delineates the blocks

How many times are these lines printed?

Do we understand?

```
>>>for variable in sequence:
>>>    block
```

```
k = 0
for i in [1, 2, 3]:
    for j in [1, 2, 3]:
        k = k + 1
print(k)
```

Is this statement important? Why

What is the value of k? Why

(Not so) Empty nest

```
>>>for variable in sequence:
>>>    block
```

- Remember that `variable` takes on the *type* of what is returned from `in`

```
sequence=[1, 2, 3]
for i in sequence:
    print(i)
```

Iteration	i
1	1
2	2
3	3

(Not so) Empty nest

```
>>>for variable in sequence:
>>>    block
```

- Remember that `variable` takes on the *type* of what is returned from `in`

```
sequence=['a', 'b', 'c']
for i in sequence:
    print(i)
```

Iteration	i
1	a
2	b
3	c