Iteration

for loops, while loops, lists

Last Time

- Intro to Python
- Due:
  - Lab 2 (last night)
  - PS2 (this morning)

Reminders

- OLI Decisions Module, over weekend
- PA 2 due Monday night
- PS 3 due Tuesday Morning

Yesterday

- Introduction to Python
- Mechanics
- Some Specifics:
    - Basic datatypes
    - Operators
    - Expressions
    - Variables
    - Functions

Data Types

Integers	`4   15110   -53      0`
Floating Point Numbers	`4.0   0.80333333333` `7.34e+014`
Strings	`"hello”   "A"    " "     ””` `'there'     '"'   '15110’`
Booleans	`True    False`

Arithmetic Expressions

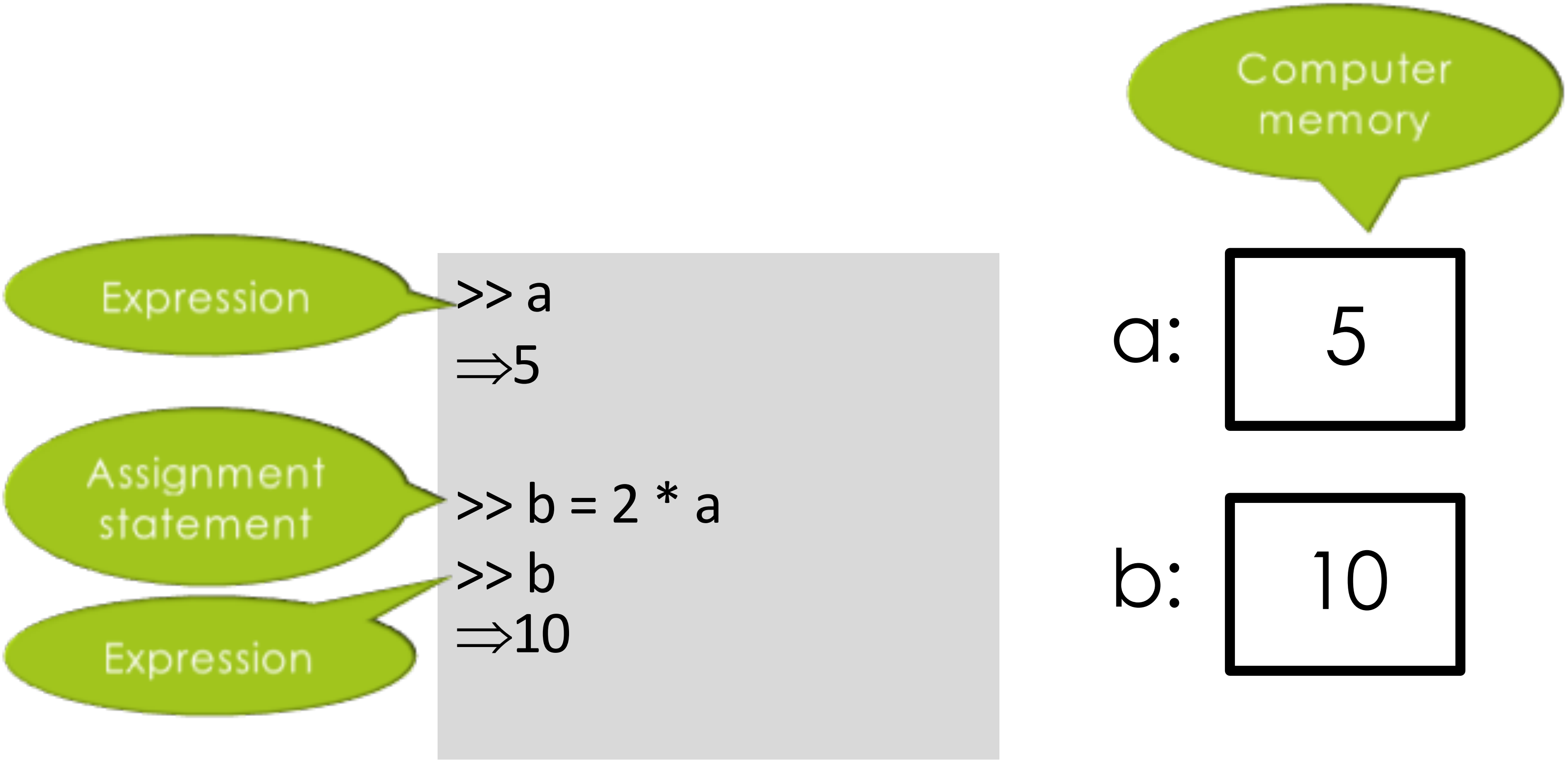
- Mathematical Operators

+	Addition		
-	Subtraction	//	Integer division
*	Multiplication	**	Exponentiation
/	Division	%	Modulo (remainder)

- Python is like a calculator: type an expression and it *evaluates the expression* (tells you the value).

```
>> 2 + 3 * 5
⇒17
```

Variables and Expressions

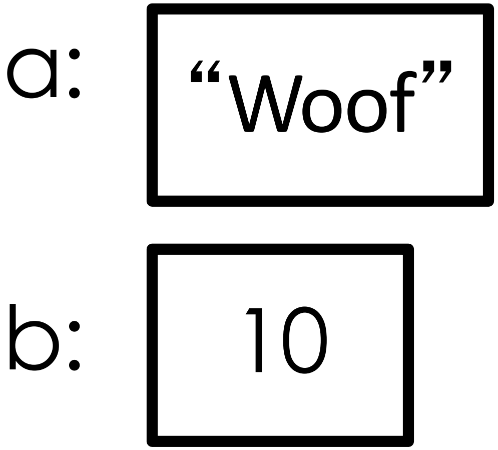


```
>> a
⇒5

>> b = 2 * a
>> b
⇒10
```

Variables

```
>> a
⇒5
>> b
⇒10
>> a = “Woof”
>> a
⇒“Woof”
>> b
⇒10
```



a: “Woof”

b: 10

Variable b does not “remember” that its value came from variable a.

Syntax vs. Semantics

Syntax

- Rules, structure
- Errors result when code is not well formed.

Semantic

- Meaning
- Error results when expression/statement can't be evaluated or executed due to meaning.

Colorless green ideas sleep furiously

Functions

- Are reusable blocks of code
- Are general
- Can be user defined and can be imported
- Are defined with parameters
- Are called with arguments

Function Syntax:

`def` *functionname*`(`*parameterlist*`)` **:**
□□□□*instructions*

Built-in Functions

Import math
r = 5 + math.sqrt(2)

Return, None, Print

```
def calculate_area(side):
   return side * side

myArea1 = calculate_area(5)



def show_area(side):
   print(side * side)

myArea2 = show_area(6)
```

Return, None, Print

```
def showAndCalc_area(side):
    area = side * side
    print(area)
    return area

myArea3 = showAndCalc_area(7)
```

End of Class problems

- Create a function that calculates 18% tip
- Input(”Enter your check’s total: ”) would return a user-entered variable. Write a short python script that would advise users of an appropriate tip based on their input.
- Create a function that takes two parameters (mass and radius) and calculates escape velocity. Note:
  - G = 6.67e-011
  - Our fine planet has mass of 5.9742e+024, and a radius of 6378.1

$$v_{\text{esc}} = \sqrt{\frac{2\,G\,M}{R}}$$

Questions?

Why do we need iteration

- Many algorithms are partially or fully a repeating set of steps.
- Can we accomplish a set of steps manually?
- Revisit the calc_tip() function – but now let's offer multiple tipping possibilities – For any check amount, let's show tips from 15% to 25%
- Try it – quick write/outline an algorithm that shows these 10 tip amounts

Creating a tip table

```
def tip_table(check):
    print(check * .15)
    print(check * .16)
    print(check * .17)
    print(check * .18)
    print(check * .19)
    print(check * .20)
    print(check * .21)
    print(check * .22)
    print(check * .23)
    print(check * .24)
    print(check * .25)
```

```
>>> tip_table(56.00)
8.4
8.96
9.520000000000001
10.08
10.64
11.200000000000001
11.76
12.32
12.88
13.44
14.0
```

Iteration

- Loops
- Provide power, generality
- Construct for iterative cycles over a range of numbers
- for x in range(y)

```
def tip_table(check):
    for tip in range(15, 25):
        print((tip * check)/100)
```

`for` Loop (simple version)

```
for loop_variable in range(n):
    loop body
```

- The loop variable is a new variable name
- The loop body is one or more instructions that you want to repeat.
- If `n` > 0, the `for` loop repeats the loop body `n` times.
- If `n` <= 0, the entire loop is skipped.
- Remember to indent loop body

`for` Loop Example

Loop variable

```
for i in range(5):
    print("hello world")
```

```
hello world

hello world

hello world

hello world

hello world
```

What happens in a loop variable?

```
for i in range(5):
  print(i)
```

```
0

1

2

3

4
```

Detour: some printing options

```
>>> for i in range(5):
...       print(i, end=" ")
0 1 2 3 4 >>>
>>>
>>> for i in range(5):
>>>       print(i, end="")
01234>>>
```

Blank space after value printed

No space after value printed

The default is `end = “\n”`.

What if we don’t want to start at zero and increase by one each time?

```
>>> for i in range(1, 6):
...     print(i, end=" ")
1 2 3 4 5 >>>
>>>
>>> for i in range(1, 6, 2):
>>>     print(i, end=" ")
1 3 5 >>>
```

Increase by 2 each time

```
range(n) gives the range 0 ... n-1
range(start, end) gives the range start ... end-1
range(start, end, step) gives the range start, start+2, ...
```

Using loop variable in arithmetic expressions

```
for i in range(10):
   print(i*2, end=" ")
```

```
0 2 4 6 8 10 12 14 16 18
```

Accumulating Outputs

building an answer a little at a time

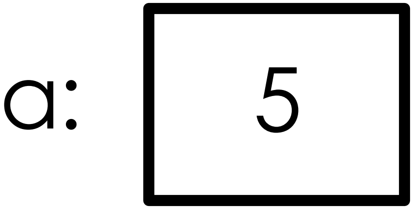
Reminder: Assignment Statements

variable = *expression*

The expression is evaluated and the result is stored in the variable

- overwrites the previous contents of *variable*.

```
>> a = 5
```



a: 5

Variables change over time

statement	value of x	value of y
`x = 150`	150	?
`y = x * 10`	150	1500
`x = x + 1`	151	1500
`y = x + y`	151	1651

Accumulating an answer

```
def sum():
  # sums first 5 positive integers
  sum = 0 # initialize accumulator
  for i in range(1, 6):
     sum = sum + i # update accumulator
  return sum # return accumulated result
```

```
>>> sum()
15
```

Now let's see what's happening under the hood

Accumulating an answer

```
def sum():
      # sums first 5 positive integers
      sum = 0 # initialize accumulator
      for i in range(1, 6):
            sum = sum + i # update accumulator
      return sum # return accumulated result
```

	i	sum
initialize sum	?	0
iteration 1	1	1
iteration 2	2	3
iteration 3	3	6
iteration 4	4	10
iteration 5	5	15

Danger! Don’t grab the loop variable!

```
for i in range(5):
    print(i, end=" ")
    i = 10
```

```
0 1 2 3 4
```

Even if you modify the loop variable in the loop, it will be reset to its next expected value in the next iteration.

```
for i in range(5):
    i = 10
    print(i, end=" ")
```

```
10 10 10 10 10
```

NEVER modify the loop variable inside a `for` loop.

Generalizing sum

```
def sum(n):
  # sums the first n positive integers
  sum = 0 # initialize
  for i in range(1, n + 1):
      sum = sum + i # update
  return sum # accumulated result
```

```
sum(6)        returns 21
sum(100)      returns 5050
sum(15110)    returns 114163605
```

Accumulation by multiplying as well as by adding

An epidemic:

```
def compute_sick(d):
    # computes total sick after d days
    newly sick = 1 # initially 1 sick person
    total_sick = 1

    for day in range(2, d + 1):
    # each iteration represents one day

        newly_sick = newly_sick * 2
        total_sick = total_sick + newly_sick

    return total_sick
```

Each newly infected person infects 2 people the next day.

Output: how an epidemic grows

```
compute_sick(1)  => 1
compute_sick(2)  => 3
compute_sick(3)  => 7
compute_sick(4)  => 15
compute_sick(5)  => 31
compute_sick(6)  => 63
compute_sick(7)  => 127
compute_sick(8)  => 255
compute_sick(9)  => 511
compute_sick(10) => 1023
compute_sick(11) => 2047
compute_sick(12) => 4095
compute_sick(13) => 8191
compute_sick(14) => 16383
compute_sick(15) => 32767
compute_sick(16) => 65535
compute_sick(17) => 131071
compute_sick(18) => 262143
compute_sick(19  => 524287
compute_sick(20) => 1048575
compute_sick(21) => 2097151
```

In just three weeks, over 2 million people are infected!

(This is what Blown To Bits means by *exponential growth*. We will see important computational problems that get exponentially “harder” as the problems gets bigger.)

Try: Create flow charts for

- Calculating interest on a savings account at 6% interest for 3 years with a starting balance of $1000.
- Generalize the above – let the user indicate the interest rate and length of time.
- Parable: grains of rice on a chessboard, (1 grain on square one, 2 grains on square 2, 4 grains on square 3 …. through square 64)

Back to our epidemic

Each newly infected person infects 2 people the next day.
The function returns the number of sick people after n days.

```
def compute_sick(d):
    # computes total sick after d days
    newly sick = 1 # initially 1 sick person
    total_sick = 1

    for day in range(2, d + 1):
    # each iteration represents one day

        newly_sick = newly_sick * 2
        total_sick = total_sick + newly_sick

    return total_sick
```

Variation on the Epidemic Example

Let us write a function that

- **Inputs the size of the population**
- **Outputs the number of days left before all the population dies out**

How can we do that using iteration (loops)?

Keep track of the number of sick people.

But do we know how many times we should loop?

Recall the Epidemic Example

```
def days_left(population):
    # computes the number of days until extinction
    days = 1
    newly_sick = 1
    total_sick = 1
    while total_sick < population:
        # each iteration represents one day
        newly_sick = newly_sick * 2
        total_sick = total_sick + newly_sick
        days = days + 1
    print(days, ” days for the population to die off")
    return days
```

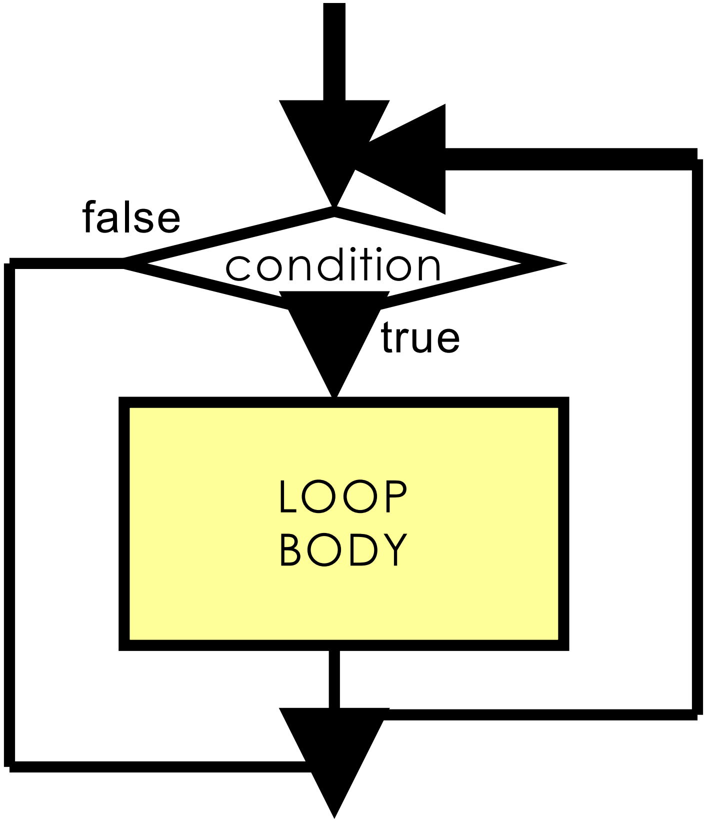
`while` loop

Format:

```
while condition:
    loop body
```

one or more instructions to be repeated

If the loop condition becomes false during the loop body, the loop body still runs to completion before we exit the loop and go on with the next step.



Recall the Epidemic Example

```
def days_left(population):
    # computes the number of days until extinction
    days = 1
    newly_sick = 1
    total_sick = 1
    while total_sick < population:
        #each iteration represents one day
        newly_sick = newly_sick * 2
        total_sick = total_sick + newly_sick
        days = days + 1
    print(days, "days for the population to die off")
    return days
```

Loop condition

While Loop Examples

Prints first 10 positive integers

```
i = 1
while  i < 11:
    print(i)
    i = i + 1
```

How about the following?

```
i = 0
while  i < 10:
    i = i + 1
    print(i)
```

What is the value of i when we exit the loop?

While vs. For Loops

```
# Prints first 10 positive integers

 i = 1
 while  i < 11:
     print(i)
     i = i + 1
```

```
# Prints first 10 positive integers

for i in range(1,11):
    print(i)
```

When to use `for` or `while` loops

- If you know in advance **how many times** you want to run a loop use a **`for`** loop.
- When you **don't know the number** of repetition needed, use a **`while`** loop.

Try: Create flow charts for

- Saving money to buy a new car – how long will it take to save for a new Tesla Model X @ $80,000. (5000.00 in a savings account)
- Saving for retirement – for different retirement targets, and calculate how long it will take to reach that target. Identify your variables and pre-assign values.
- Can you generalize the above to accommodate different user input?