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
  - `*` Multiplication
  - `/` Division
  - `%` Modulo (remainder)
  - `//` Integer division
  - `**` Exponentiation

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

\[
\text{Example: } 2 + 3 * 5 \Rightarrow 17
\]
Variables and Expressions

Expression

Assignment statement

Expression

```plaintext
>> a
⇒ 5

>> b = 2 * a
>> b
⇒ 10
```

Computer memory

a: 5

b: 10
Variables

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

```
>> 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:

```python
def functionname (parameterlist) :
    instructions
```

Built-in Functions

Import math

$$r = 5 + \text{math.sqrt}(2)$$
def calculate_area(side):
    return side * side

myArea1 = calculate_area(5)

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

myArea2 = show_area(6)
def showAndCalc_area(side):
    area = side * side
    print(area)
    return area

myArea3 = showAndCalc_area(7)
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:

- $G = 6.67 \times 10^{-11}$
- Our fine planet has mass of $5.9742 \times 10^{24}$, and a radius of $6378.1$
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)

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

```python
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 \( n \leq 0 \), the entire loop is skipped.
- Remember to indent loop body.
for Loop Example

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

hello world
hello world
hello world
hello world
hello world
What happens in a loop variable?

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

0
1
2
3
4
Detour: some printing options

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

```python
>>> for i in range(5):
>>>     print(i, end="")
01234>>>  
```

The default is `end = "\n"`. 

Blank space after value printed

No space after value printed
What if we don’t want to start at zero and increase by one each time?

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

```python
>>> 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

```python
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

\[ \text{variable} = \text{expression} \]

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

- overwrites the previous contents of variable.

\[ \text{>> a} = 5 \quad \text{a:} \quad 5 \]
Variables change over time

<table>
<thead>
<tr>
<th>statement</th>
<th>value of x</th>
<th>value of y</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 150</td>
<td>150</td>
<td>?</td>
</tr>
<tr>
<td>y = x * 10</td>
<td>150</td>
<td>1500</td>
</tr>
<tr>
<td>x = x + 1</td>
<td>151</td>
<td>1500</td>
</tr>
<tr>
<td>y = x + y</td>
<td>151</td>
<td>1651</td>
</tr>
</tbody>
</table>
Accumulating an answer

```python
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.
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

<table>
<thead>
<tr>
<th>i</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial sum</td>
<td>0</td>
</tr>
<tr>
<td>iteration 1</td>
<td>1</td>
</tr>
<tr>
<td>iteration 2</td>
<td>3</td>
</tr>
<tr>
<td>iteration 3</td>
<td>6</td>
</tr>
<tr>
<td>iteration 4</td>
<td>10</td>
</tr>
<tr>
<td>iteration 5</td>
<td>15</td>
</tr>
</tbody>
</table>
Danger! Don’t grab the loop variable!

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

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

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

NEVER modify the loop variable inside a for loop.
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
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

Accumulation by multiplying as well as by adding

An epidemic:

Each newly infected person infects 2 people the next day.
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?
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:

```python
while condition:
    loop body
```

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.
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
# Prints first 10 positive integers

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

How about the following?

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

What is the value of $i$ when we exit the loop?
# Prints first 10 positive integers

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

# Prints first 10 positive integers

```python
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?