

# UNIT 2B

## An Introduction to Programming ("for"-loops)

# Announcements

- Programming assignment 2 is due Thursday night at 11:59 PM
- Is there anybody who needs help for setting up laptops?

# This Lecture

- Review of Last Lecture
- A control structure for iteration (repetition):  
“for loops”
- More Python (python3) practice

# Last Lecture

- Basic data types
- Variables
- Expressions
- Assignment statements
- Methods (functions)
- Modules such as math

# Functions

- Built-in functions
- User defined functions
  - With parameter
  - Without parameter
  - With returning a value
  - Without returning a value
- Modularity
- Naming, commenting, testing

# Practice Questions at Piazza

```
def calculateAverage_1(num1, num2):  
    print("Average of ", num1, " and ", num2, " is ", (num1 + num2)/2)  
  
def calculateAverage_2(num1, num2):  
    average = (num1 + num2) / 2  
    print("Average of ", num1, " and ", num2, " is ", average)  
  
def calculateAverage_3(num1, num2):  
    sum = num1 + num2  
    print("Average of ", num1, " and ", num2, " is ", sum / 2)  
  
def averageOf(num1, num2):  
    sum = num1 + num2  
    return(sum / 2)  
  
def calculateAverage_4(num1, num2):  
    print ("Average of ", num1, " and ", num2, " is ", averageOf(num1, num2))
```

# Practice

- Write a function to print average of 30, 67 and 89

```
def averageOf_30_67_89():  
    return (30+67+89)/3
```

- Write a function to print average of 3 given numbers

```
def averageOf3Num(num1, num2, num3):  
    return (num1+num2+num3)/3
```

# Functions and Generalization

```
def tipFor(amount) :  
    return amount * .18
```

Can we make it more  
flexible and general

```
>>> tipFor(25)  
4.5
```

```
def tipFor(amount, percentage) :  
    return amount * percentage
```

```
>>> tipFor(25, 0.18)  
4.5  
>>> tipFor(25, 0.10)  
2.5
```



# Functions and Generalization

```
def multiply(num1, num2):  
    return num1 * num2
```

```
def tipFor(amount, percentage):  
    return amount * percentage
```


If we have a multiply function, do we need to have both?

**YES.** It helps readability, and makes thinking much easier.

# Functions and Generalization

```
def tipFor(amount, percentage):  
    return amount * percentage
```

```
>>> tipFor(25, 0.18)  
4.5
```



**Which one? Why?**

```
def tipFor(amount, percentage):  
    return amount * percentage / 100
```

```
>>> tipFor(25, 18)  
4.5
```

small programs, big outputs

# USING ITERATION

# Why Iteration?

- More generality so more power
- Example: remember the tip method:

```
def tipFor(amount, percentage):  
    return amount * percentage
```

```
>>> tip(25, 0.18)  
4.5
```

But what if we want a table of tip amounts?

# Table of Tips (the hard way)

```
def tip_table_1() :  
    print(tipFor(10, 0.18))  
    print(tipFor(11, 0.18))  
    print(tipFor(12, 0.18))  
    print(tipFor(13, 0.18))  
    # etc. for more values
```

```
>>> tip_table_1()  
1.7999999999999998  
1.98  
2.16  
2.34
```

# Table of Tips (the easy way)

```
def tip_table_2(low, high) :  
    for amount in range(low, high+1):  
        print(tipFor(amount, 0.18))
```

```
>>> tip_table_2(10, 20)  
1.7999999999999998  
1.98  
2.16  
2.34  
2.52  
2.6999999999999997  
2.88  
3.06  
3.2399999999999998  
3.42  
3.5999999999999996  
>>>
```

# 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  $n \leq 0$ , the entire loop is skipped.
- Remember to **indent** loop body

# for Loop Example

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

Loop control variable

hello world

hello world

hello world

hello world

hello world



# What Happens to 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 >>>
```

Blank space after value printed

```
>>>
```

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

```
>>>     print(i, end="")
```

```
01234>>>
```

No space after value printed

# 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



# Using loop variable in arithmetic expressions

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

0 2 4 6 8 10 12 14 16 18

**Note that we can give different names to loop variable. i, j, k can generally be accepted. While naming, pay attention to the context.**

building an answer a little at a time

# ACCUMULATOR LOOPS

# 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
$y = y + 1$	150	1501
$x = x + y$	1651	1501

# Accumulating an answer

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

	number	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 sum  
    for i in range(1, n + 1) :  
        sum = sum + i #update  
    return sum #accumulated result
```

```
sum(6)          => 21  
sum(100)        => 5050  
sum(15110)      => 114163605
```

# Accumulation by multiplying as well as by adding

An epidemic:

```
def compute_infected(n):  
    #computes total infected after n days  
    newly_infected = 1  
    total_infected = 1  
    for day in range(2, n + 1):  
        #each iteration represents one day  
        newly_infected = newly_infected * 2  
        total_infected = total_infected + newly_infected  
    return total_infected
```



Each newly infected person  
infects 2 people the next day.

# Output: how an epidemic grows

```
compute_infected(1)  => 1
compute_infected(2)  => 3
compute_infected(3)  => 7
compute_infected(4)  => 15
compute_infected(5)  => 31
compute_infected(6)  => 63
compute_infected(7)  => 127
compute_infected(8)  => 255
compute_infected(9)  => 511
compute_infected(10) => 1023
compute_infected(11) => 2047
compute_infected(12) => 4095
compute_infected(13) => 8191
compute_infected(14) => 16383
compute_infected(15) => 32767
compute_infected(16) => 65535
```

```
compute_infected(17)  => 131071
compute_infected(18)  => 262143
compute_infected(19)  => 524287
compute_infected(20)  => 1048575
compute_infected(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.)

working backwards

# DECREASING THE LOOP VARIABLE

# Countdown! (the easy way)

```
import time
def countdown():
    for i in range(10, 0, -1):
        print(i)
        time.sleep(1) #pauses for 1 sec.
```

Now `i` gets smaller at each iteration.



```
countdown()
```

```
⇒ 10
```

```
⇒ 9
```

```
⇒ 8
```

```
⇒ ...
```

```
⇒ 1
```

# Summary

- `for` loops **repeat** a computation a **specified number of times**
- The **loop variable** is **automatically increased** or decreased for each iteration
- We can **accumulate** an answer by repeated addition or multiplication of an accumulator variable
- We can use `end=` **with** `print`



# Next Week

- New concept: **algorithm**
- New control structures
  - While loops
  - Conditionals