

Iteration

for loops, while loops



Last Time

- Intro to Python

- Due:

 - Academic Integrity Form (now!)

 - PS2 (this morning, 9:00AM)

 - Lab 2 (July 3rd)

 - OLI Module on Iteration (July 3rd, 11:59PM)

Reminders

- ▣ OLI Decisions Module, over weekend
- ▣ PA 2 due Monday night (July 8th)
- ▣ Lab on Monday: Review!
 - ▣ Monday, July 8th at 4:30 in GHC 5207

Reminders

- Use the PS Templates!
- I have OH today; any issues with OH?

Review from last time

Review from last time

- ▣ Introduction to Python
- ▣ Mechanics
- ▣ Some Specifics:
 - ▣ Programming Languages
 - ▣ Basic datatypes
 - ▣ Variables
 - ▣ Functions

Data Types

Integers

```
4 15110 -53 0
```

Floating Point Numbers

```
4.0 0.8033333333333333  
7.34e+014
```

Strings

```
"hello" "A" " " "" ""  
'there' '' '15110'
```

Booleans

```
True False
```

Literal None

```
None
```

Arithmetic Expressions

- Mathematical Operators

+ Addition

- Subtraction

* Multiplication

/ Division

//

**

%

Integer division

Exponentiation

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
```

```
>>> a
```

```
⇒ 5
```

Expression

```
>>> a
```

```
⇒ 5
```

Assignment statement

```
>>> b = 2 * a
```

```
>>> b
```

Expression

```
⇒ 10
```

Computer memory

a:

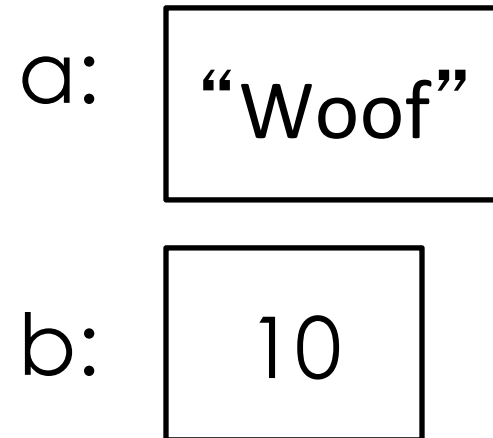
5

b:

10

Variables

```
>> a  
=>5  
>> b  
=>10  
>> a = "Woof"  
>> a  
=>"Woof"  
>> b  
=>10
```



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

Functions

- ▣ Are reusable blocks of code
- ▣ Are general
- ▣ Can be user defined
- ▣ 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)
```

Defining vs Calling a Function

Defining a function

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

```
def multiply(v1, v2, v3) :  
    return v1*v2*v3
```

```
def hello_world():  
    print("Hello World")
```

Calling a function

```
>>> functionname (argumentlist)
```

```
>>> multiply(5, 2, 3)  
30
```

```
>>> hello_world()  
Hello World
```

Return, None, Print

Function **returns value**

```
def multiply(v1, v2, v3):  
    return v1 * v2 * v3
```

```
myMulti_1= multiply(3,6,7)
```

```
>>> type(myMulti_1)
```

```
<class 'int'>
```

Function prints value and **returns None** (by default)

```
def multiply(v1, v2, v3):  
    print(v1 * v2 * v3)
```

```
myMulti_2= multiply(3,6,7)
```

```
>>> type(myMulti_2)
```

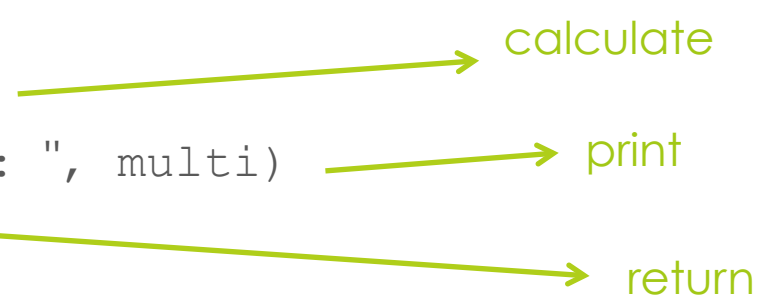
```
<class 'NoneType'>
```

Return, None, Print

Function calculates, prints, returns value

```
def multiply(v1,v2,v3):  
    multi = v1 * v2 * v3  
    print("The result is: ", multi)  
    return multi
```

myMulti = multiply(3,6,7)



```
>>> type(myMulti)
```

```
<class 'int'>
```

End of Class problems

- Create a function that calculates 18% tip
- `input("Enter your total: bill")` would return a user-entered variable. Write a short python script that would advise users of an appropriate tip based on their input.

End of Class problems

- Create a function that calculates 18% tip
- `input("Enter your total: bill")` would return a user-entered variable. Write a short python script that would advise users of an appropriate tip based on their input

```
1 def tip_calculator(bill):
2     tip = bill*18/100
3     final_pay = bill + tip
4     return final_pay
5
6 #The input function returns a string!!
7 user_bill_input = input("Enter your bill: ")
8
9 print("Input is a: ", type(user_bill_input))
10
11 #Cast (convert) the string to number with the
12 int function
13 cast_user_bill_input = int(user_bill_input)
14
15 print("Your final bill including tip: ",
16       tip_calculator(cast_user_bill_input))
```

```
[GCC 4.8.2] on linux
Enter your bill: 100
Input is a: <class 'str'>
Your final bill including tip: 118.0
>
```


End of Class problems

- 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

$$v_{\text{esc}} = \sqrt{\frac{2GM}{R}}$$

End of Class problems

```
1 import math
2
3 def compute_ev(M, R):
4     # computes escape velocity
5     G = 6.67e-011
6     return math.sqrt(2*G*M/R)
7
8 earth_ev = compute_ev(5.9742e+024,
9 6378.1)
10 print("Earth's escape velocity is:
    ", earth_ev)
```

```
[GCC 4.8.2] on Linux
Earth's escape velocity is: 353485.929578262
```

$$v_{\text{esc}} = \sqrt{\frac{2GM}{R}}$$

Questions?

Iteration

Why do we need iteration

- Many algorithms are partially or fully a repeating set of steps.
- Can we accomplish a set of steps manually?
- Let's say you want to tip the waiter but you are not sure how much. The tipping possibilities you consider are from **15% to 25%**
- How would you do this?

Creating a tip table

```
check = int(input("Enter 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)
```

```
>>> Enter check:57
8.549999999999999
9.1200000000000001
9.6900000000000001
10.26
10.83
11.4
11.969999999999999
12.5400000000000001
13.1100000000000001
13.68
14.25
```

Iteration

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

```
check = int(input("Enter check:"))  
  
for tip in range(15, 25):  
    print((tip * check)/100)
```

For Loop Syntax

`for` is a reserved word and cannot be used as a variable name



```
for loopvariable in range(start, end, step) :  
□□□□ loop_body
```


For Loop Syntax

`for` is a reserved word and cannot be used as a variable name

loop variable is a new variable name



```
for loopvariable in range(start, end, step) :  
□□□□ loop_body
```

For Loop Syntax

`for` is a reserved word and cannot be used as a variable name

gives the range
`start, start+step ... end-1`

loop variable is a new variable name

```
for loopvariable in range(start, end, step) :  
□□□□ loop_body
```

For Loop Syntax

`for` is a reserved word and cannot be used as a variable name

gives the range
`start, start+step ... end-1`

loop variable is a new variable name

```
for loopvariable in range(start, end, step) :  
□□□□ loop_body
```

declares the start of an indented block

For Loop Syntax

`for` is a reserved word and cannot be used as a variable name

gives the range
`start, start+step ... end-1`

loop variable is a new variable name

```
for loopvariable in range(start, end, step) :  
□□□□ loop_body
```

Indentation is critical.
Use spaces *only*, **not tabs!**

declares the start of an indented block

For Loop Syntax

`for` is a reserved word and cannot be used as a variable name

gives the range
`start, start+step ... end-1`

loop variable is a new variable name

```
for loopvariable in range(start, end, step) :  
□□□□ loop_body
```

One or more instructions that you want to repeat

declares the start of an indented block

Indentation is critical.
Use spaces *only*, **not tabs!**

for Loop Example

Loop variable

gives the range
0, 1, 2, 3, 4

```
for i in range(0, 5, 1):  
    print("hello world")
```

hello world

hello world

hello world

hello world

hello world

Loop body
(gets repeated 5 times)

What happens in a loop variable?

```
for i in range(0, 5, 1):  
    print("hello world, i=", i)
```

hello world, i=0 ← Start = 0

hello world, i=1

hello world, i=2 ← Increment by 1

hello world, i=3

hello world, i=4 ← Ends when reaches 4
(up to *end-1*)

for Loop Example (changing start and end)

```
for i in range(0, 8, 1):  
    print(i)
```

0

1

2

3

4

5

6

7

```
for i in range(12, 15, 1):  
    print(i)
```

12

13

14

for Loop Example (changing increment)

```
for i in range(0, 8, 1):  
    print(i)
```

0
1
2
3
4
5
6
7

```
for i in range(0, 8, 3):  
    print(i)
```

0
3
6

```
for i in range(10, 100, 25):  
    print(i)
```

10
35
60
85

for Loop Example (decrement)

```
for i in range(10, 0, -2):  
    print(i)
```

10

8

6

4

2

What will happen?

Ex #1

```
for i in range(0, 10, -1):  
    print(i)
```

Ex #2

```
for i in range(10, 0, 1):  
    print(i)
```

What will happen?

Ex #1

```
for i in range(0, 10, -1):  
    print(i)
```

Ex #2

```
for i in range(10, 0, 1):  
    print(i)
```

Loop will be skipped!!

Other versions of the range function

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

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

`range(start, end)` gives the range start ... end-1

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

`range(n)` gives the range 0 ... n-1

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

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

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(0, 5):
```

```
i = 10
```



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

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



```
10 10 10 10 10
```

Accumulating Outputs

building an answer a little at a time

Variables and Expressions

```
>>> a = 5
```

```
>>> a
```

```
⇒ 5
```

Expression

```
>>> a
```

```
⇒ 5
```

Assignment statement

```
>>> b = 2 * a
```

```
>>> b
```

Expression

```
⇒ 10
```

Computer memory

a:

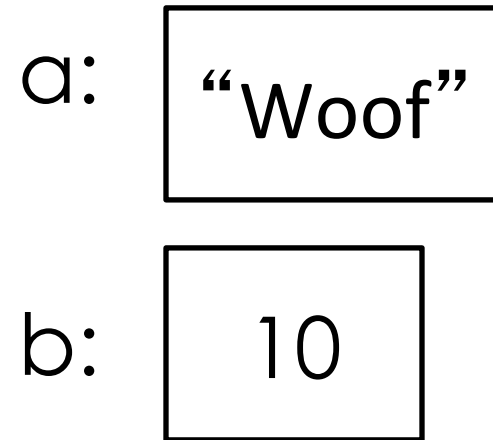
5

b:

10

Variables

```
>> a  
⇒5  
>> b  
⇒10  
>> a = "Woof"  
>> a  
⇒"Woof"  
>> b  
⇒10
```



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

Variables change over time

in Python

value of x in memory

value of y in memory

```
x = 150
```

150

(not yet defined)

```
y = x * 10
```

150

1500

```
x = x + 1
```

151

1500

```
y = x + y
```

151

1651

Variables change over time

in Python

value of x in memory

value of y in memory

```
x = 150
```

```
y = x * 10
```

```
x = x + 1
```

```
y = x + y
```

150

150

151

151

(not yet defined)

1500

1500

1651

Variables change over time

in Python

value of x in memory

value of y in memory

```
x = 150
```

```
y = x * 10
```

```
x = x + 1
```

```
y = x + y
```

150

150

151

151

(not yet defined)

1500

1500

1651



Variables change over time

in Python

```
x = 150  
  
y = x * 10  
  
x = x + 1  
  
y = x + y
```

value of x in memory

150

150

151

151

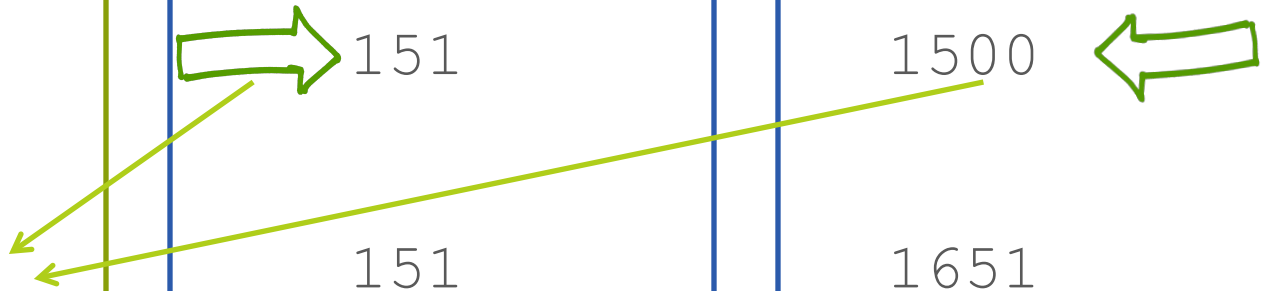
value of y in memory

(not yet defined)

1500

1500

1651



Accumulating an answer

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

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

```
15
```

Accumulating an answer

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

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

```
15
```

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

Accumulating an answer

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

Generalizing sum

*# sums the first **n** positive integers*

sum(6) returns 21

sum(100) returns 5050

sum(15110) returns 114163605

Generalizing sum

```
# sums the first n positive integers
```

```
def sum(n):
```

```
    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
```

An Epidemic

Accumulation by multiplying as well as by adding

An epidemic:

Each newly infected person infects 2 people the next day.

```
1 # computes total sick after d days
2 def compute_sick(d):
3     newly_sick = 1 # initially 1 sick person
4     total_sick = 1
5     for day in range(2, d + 1):
6         # each iteration represents one day
7         newly_sick = newly_sick * 2
8         total_sick = total_sick + newly_sick
9     return total_sick
```

Accumulation by multiplying as well as by adding

An epidemic:

Each newly infected person infects 2 people the next day.

```
1  # computes total sick after d days
2  def compute_sick(d):
3      newly_sick = 1 # initially 1 sick person
4      total_sick = 1
5      for day in range(2, d + 1):
6          # each iteration represents one day
7          newly_sick = newly_sick * 2
8          total_sick = total_sick + newly_sick
9      return total_sick
```

Accumulation by multiplying as well as by adding

An epidemic:

Each newly infected person infects 2 people the next day.

```
1  # computes total sick after d days
2  def compute_sick(d):
3      newly_sick = 1 # initially 1 sick person
4      total_sick = 1
5      for day in range(2, d + 1):
6          # each iteration represents one day
7          newly_sick = newly_sick * 2
8          total_sick = total_sick + newly_sick
9      return total_sick
```

Accumulation by multiplying as well as by adding

An epidemic:

Each newly infected person infects 2 people the next day.



```
1 # computes total sick after d days
2 def compute_sick(d):
3     newly_sick = 1 # initially 1 sick person
4     total_sick = 1
5     for day in range(2, d + 1):
6         # each iteration represents one day
7         newly_sick = newly_sick * 2
8         total_sick = total_sick + newly_sick
9     return total_sick
```


Accumulation by multiplying as well as by adding

An epidemic:

Each newly infected person infects 2 people the next day.



```
1  # computes total sick after d days
2  def compute_sick(d):
3      newly_sick = 1 # initially 1 sick person
4      total_sick = 1
5      for day in range(2, d + 1):
6          # each iteration represents one day
7          newly_sick = newly_sick * 2
8          total_sick = total_sick + newly_sick
9      return total_sick
```

Accumulation by multiplying as well as by adding

An epidemic:

Each newly infected person infects 2 people the next day.

```
1  # computes total sick after d days
2  def compute_sick(d):
3      newly_sick = 1 # initially 1 sick person
4      total_sick = 1
5      for day in range(2, d + 1):
6          # each iteration represents one day
7          newly_sick = newly_sick * 2
8          total_sick = total_sick + newly_sick
9      return total_sick
```

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!

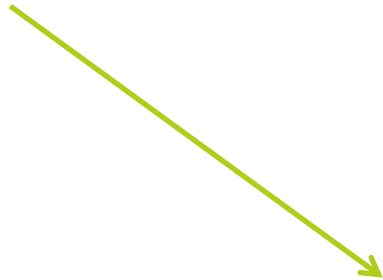
(This is what Blown To Bits
means by *exponential growth*.)

We will see important
computational problems that
get exponentially “harder” as
the problems gets bigger.)

While Loops

While Loop Syntax

`while` is a reserved word and cannot be used as a variable name



```
while condition:  
    □□□□ loop_body
```

While Loop Syntax

`while` is a reserved word and cannot be used as a variable name

The *loop_body* executes while the *condition* holds true



```
while condition:  
    □□□□ loop_body
```

While Loop Syntax

`while` is a reserved word and cannot be used as a variable name

The *loop_body* executes while the *condition* holds true

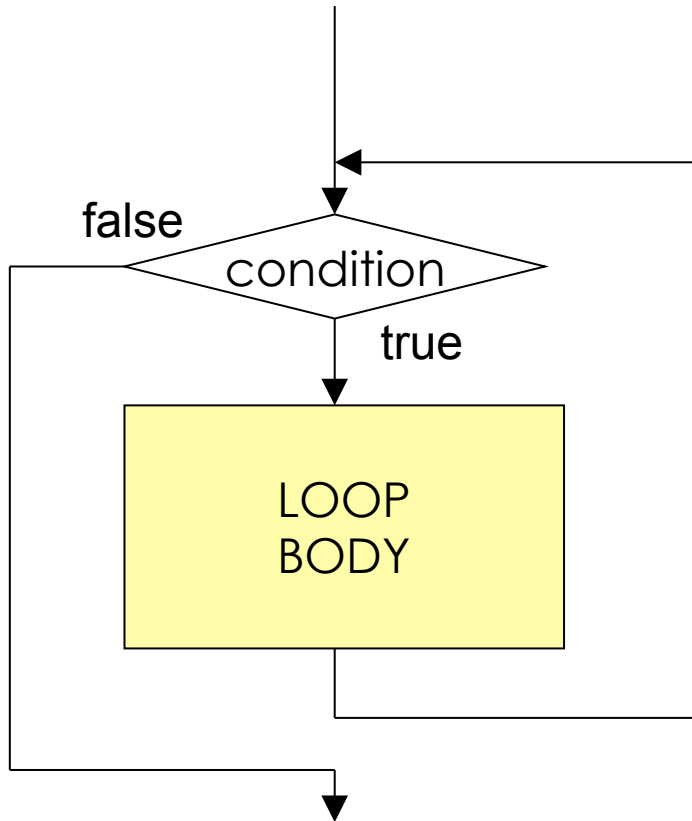
```
while condition:  
    □□□□ loop_body
```

Indentation is critical.
Use spaces *only*, **not tabs!**

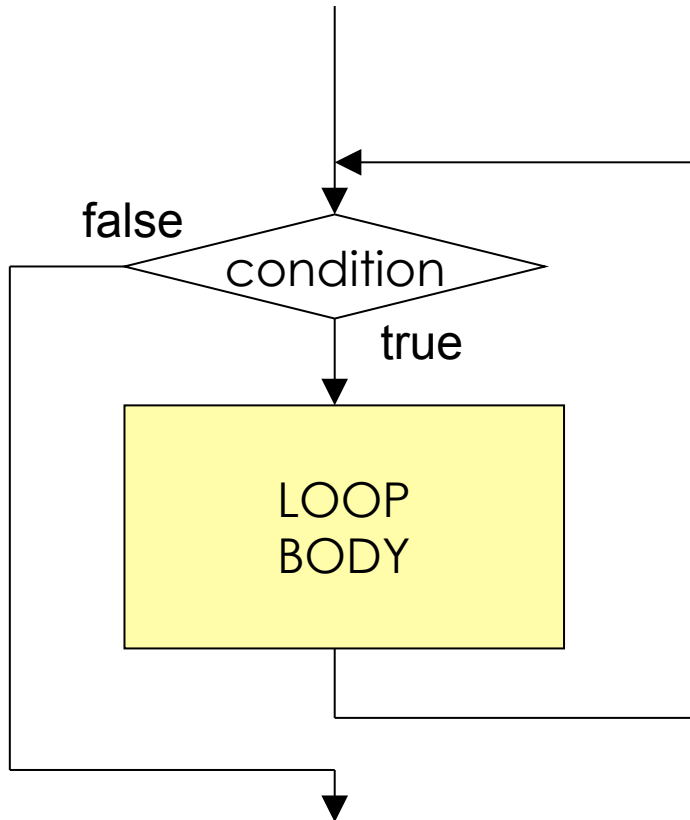
One or more instructions that you want to repeat

declares the start of an indented block

while loop



while loop



NOTE: 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.

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?

Be careful of infinite loops!!

Infinite loop:

Condition always true because loop variable is never changed in the body of the loop!

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

Infinite loop:

Condition always true!

```
i = 10
while True:
    i = i + 1
    print(i)
```

```
i = 10
while 1 < 5:
    i = i + 1
    print(i)
```

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.

```
1  # computes total sick after d days
2  def compute_sick(d):
3      newly_sick = 1 # initially 1 sick person
4      total_sick = 1
5      for day in range(2, d + 1):
6          # each iteration represents one day
7          newly_sick = newly_sick * 2
8          total_sick = total_sick + newly_sick
9      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

```
1 # computes the number of days until extinction
2 def days_left(population):
3     days = 1
4     newly_sick = 1
5     total_sick = 1
6     while total_sick < population:
7         # each iteration represents one day
8         newly_sick = newly_sick * 2
9         total_sick = total_sick + newly_sick
10        days = days + 1
11    print(days, "days for the population to die off")
12    return days
13
```

Recall the Epidemic Example

```
1 # computes the number of days until extinction
2 def days_left(population):
3     days = 1
4     newly_sick = 1
5     total_sick = 1
6     while total_sick < population:
7         # each iteration represents one day
8         newly_sick = newly_sick * 2
9         total_sick = total_sick + newly_sick
10        days = days + 1
11        print(days, "days for the population to die off")
12    return days
13
```

Recall the Epidemic Example

```
1 # computes the number of days until extinction
2 def days_left(population):
3     days = 1
4     newly_sick = 1
5     total_sick = 1
6     while total_sick < population:
7         # each iteration represents one day
8         newly_sick = newly_sick * 2
9         total_sick = total_sick + newly_sick
10        days = days + 1
11        print(days, "days for the population to die off")
12    return days
13
```

Loop condition

Recall the Epidemic Example

```
1  # computes the number of days until extinction
2  def days_left(population):
3      days = 1
4      newly_sick = 1
5      total_sick = 1
6      while total_sick < population:
7          # each iteration represents one day
8          newly_sick = newly_sick * 2
9          total_sick = total_sick + newly_sick
10         days = days + 1
11         print(days, "days for the population to die off")
12     return days
13
```

Recall the Epidemic Example

```
1 # computes the number of days until extinction
2 def days_left(population):
3     days = 1
4     newly_sick = 1
5     total_sick = 1
6     while total_sick < population:
7         # each iteration represents one day
8         newly_sick = newly_sick * 2
9         total_sick = total_sick + newly_sick
10        days = days + 1
11    print(days, "days for the population to die off")
12    return days
13
```

Recall the Epidemic Example

```
1  # computes the number of days until extinction
2  def days_left(population):
3      days = 1
4      newly_sick = 1
5      total_sick = 1
6      while total_sick < population:
7          # each iteration represents one day
8          newly_sick = newly_sick * 2
9          total_sick = total_sick + newly_sick
10         days = days + 1
11         print(days, "days for the population to die off")
12     return days
13
```

For vs While 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)
```

While vs. For Loops

Prints first 5 even integers

```
i = 2
while i < 11:
    print(i)
    i = i + 2
```

Prints first 5 even integers

```
for i in range(2, 11, 2):
    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

- 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)

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?