Objectives

- Boolean operators
- Indefinite Loops

Review

- Former general design pattern:
  1. Optionally, get user input
  2. Do some computation
  3. Display results
- Now general design pattern:
  1. Optionally, get user input
  2. Do some computation in **functions**, get results
  3. Display results

More Complex Conditions

- Boolean
  - Two logical values: True and False
- Combine conditions with Boolean operators
  - **and** – True only if **both** operands are True
  - **or** – True if **at least one** operand is True
  - **not** – True if the operand is not True
- English examples
  - If it is raining **and** it is cold
  - If it is Saturday **or** it is Sunday
  - If the shirt is on sale **or** the shirt is purple

What is the output?

Focus: how operations work
Not good variable names

```
x = 2
y = 3
z = 4

b = x==2
c = not b
d = (y<4) and (z<3)
print("d=",d)
d = (y<4) or (z<3)
print("d=",d)

d = not d
print(b, c, d)
```

Because of precedence, we don't *need* parentheses

Truth Tables

operands

A	B	A and B	A or B	not A	not B	not A and B	A or not B
T	T						
T	F						
F	T						
F	F						

Truth Tables

operands

A	B	A and B	A or B	not A	notB	not A and B	A or not B
T	T	T	T				
T	F	F	T				
F	T	F	T				
F	F	F	F				

Truth Tables

operands

A	B	A and B	A or B	not A	notB	not A and B	A or not B
T	T	T	T	F	F		
T	F	F	T	F	T		
F	T	F	T	T	F		
F	F	F	F	T	T		

Truth Tables

operands

A	B	A and B	A or B	not A	notB	not A and B	A or not B
T	T	T	T	F	F	F	T
T	F	F	T	F	T	F	T
F	T	F	T	T	F	T	F
F	F	F	F	T	T	F	T

Practice: Numeric Grade Input Range

- Enforce that user must input a numeric grade between 0 and 100
  - In Python, we can't (always) write a condition like 0 <= num_grade <= 100, so we need to break it into two conditions
- Write an appropriate condition for this check on the numeric grade
  - Using **and**
  - Using **or**

Practice: Numeric Grade Input Range

- Enforce that user must input a numeric grade between 0 and 100
  - Using **and**

```
if num_grade >= 0 and num_grade <= 100:
    computation
else:
    print error message
```

  - Using **or**

```
if num_grade < 0 or num_grade > 100:
    print error message
else:
    computation
```

Short-circuit Evaluation

- Don't necessarily need to evaluate all expressions in a compound expression
- A `and` B
  - If A is `False`, compound expression is `False`
- A `or` B
  - If A is `True`, compound expression is `True`
- No need to evaluate B
  - Put more important/limiting expression first
  - Example:

```
if count != 0 and sum/count > 10:
    do something
```


INDEFINITE LOOPS

Indefinite Loops

- `for` loops are ***definite*** loops
  - Execute a *fixed* number of times

- ***Indefinite*** loops: keep iterating until certain conditions are met
  - Depending on condition, no guarantee in advance of how many times the loop body will be executed

While Loop Syntax

```
while condition :
    statement1
    statement2
    …
    statementn
```

loop stops when condition is False

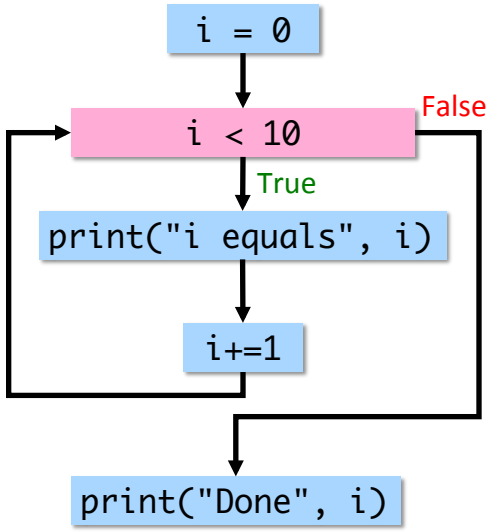
keyword

body of while loop

- Like a looped `if` statement
  - Execute statements **only** when condition is true

While Loop

```
i = 0
while i < 10 :
    print("i equals", i)
    i+=1
print("Done", i)
```



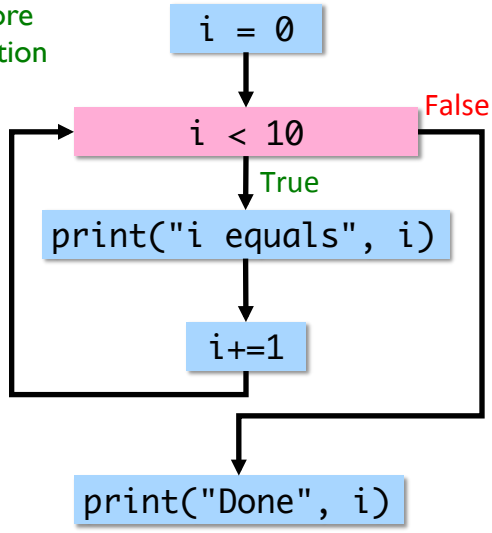
Questions:

- How many times will `i` get printed out?
- How many times is the condition evaluated?
- What is the value of `i` after the loop?

While Loop

Initialize i before using in condition

```
i = 0
while i < 10 :
    print("i equals", i)
    i+=1
print("Done", i)
```



Questions:

- How many times will `i` get printed out?
- How many times is the condition evaluated?
- What is the value of `i` after the loop?

While vs. For Loops

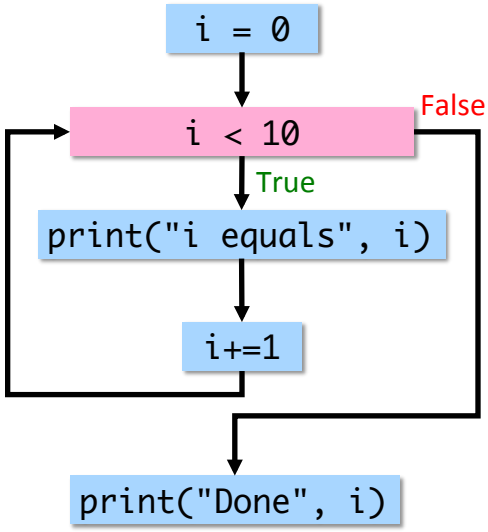
- Any `for` loop can be translated into a `while` loop
- But **NOT** vice versa

➢ `while` loops are more **powerful** than `for` loops

Convert to a `for` loop

We ***can*** convert this `while` loop into a `for` loop because it executes a ***fixed*** number of times.

```
i = 0
while i < 10 :
    print("i equals", i)
    i+=1
print("Done", i)
```



Comparing `while` and `for`

- What are the main differences between these loops?
- What are the advantages and disadvantages of each?

```
i = 0
while i < 10 :
    print("i equals", i)
    i+=1
print("Done", i)
```

```
for i in range(10):
    print("i equals", i)

print("Done", i+1)
```


What Will This Loop Do?

```
count = 1
while count > 0:
    print(count)
    count += 1
```

Infinite Loop

- Condition will never be `False` so keeps executing

```
count = 1
while count > 0:
    print(count)
    count += 1
```

- To stop an executing program in Linux use
  - Control-C

Infinite Loop Discussion

- Is there ever a time that an infinite loop is wanted?
  - Yes! For example in web servers, we have something like

```
while True:
    listenForRequest()
    handleRequest()
```

- Can a computer automatically detect infinite loops?
  - No that is an **undecidable** problem
  - Best to **prevent** infinite loops (more later)
    - Benefit of `for` loops: definite loops

A Very Simple Therapist

- Whenever a user tells the computer/program what they think, the program asks, "How does that make you feel?"
- Ends when user enters nothing ("")
- Partial example output:

```
Tell me what is bothering you.
There is too much going on in my life.
How does that make you feel?
I feel like I am out of control and can't juggle it all.
How does that make you feel?
Really stressed and tired.
How does that make you feel?

Thank you!  Come again!
```


Design Pattern: Sentinel Loop

- Sentinel: when to stop
  - "guard" to the loop

```
value = get input
while value != sentinel :
    process value
    value = get input
```

`while` Loops: comparing use of `break`

```
# condition says when loop
# will continue
x=eval(input("Enter number:"))
while x % 2 != 0 :
   print("Error!")
   x = eval(input("Enter
              number: "))
print(x, "is an even number.")
```

Says when to keep going

```
# have to look inside loop to
# know when it stops
while True :
   x = eval(input("Enter number:"))
   if x % 2 == 0 :
      break     "breaks" out of a loop
   print("Error!")
print(x, "is an even number.")
```

Says when to stop

Using break statements:
Best when loop has to execute at least once.

Summary: While vs. For Loops

- Any `for` loop can be translated into a `while` loop
- But **not** vice versa

- `while` loops are more **powerful** than `for` loops
  - Give an example of a `while` loop that can't be converted to a `for`

Looking Ahead

- Lab 5 due Friday
- Broader Issue: Self-driving Cars