INTRODUCTION TO PYTHON: DAY TWO

STEPHANIE SPIELMAN

BIG DATA IN BIOLOGY SUMMER SCHOOL, 2015
CENTER FOR COMPUTATIONAL BIOLOGY AND BIOINFORMATICS
UNIVERSITY OF TEXAS AT AUSTIN

ANOTHER DATA TYPE: DICTIONARIES

- **Dictionaries are defined with braces: `{}`**
  - Contain `key:value` pairs
  - Known in other contexts as "associative arrays"

ANOTHER DATA TYPE: DICTIONARIES

- **Dictionaries are defined with braces: {}**
  - Contain `key:value` pairs
  - Known in other contexts as "associative arrays"

```
# Define a dictionary of names
names = {"Stephanie": "Spielman", "Eleisha": "Jackson",
    "Claus": "Wilke"}
```

ANOTHER DATA TYPE: DICTIONARIES

- **Dictionaries are defined with braces: {}**
  - Contain `key:value` pairs
  - Known in other contexts as "associative arrays"

```
# Define a dictionary of names
names = {"Stephanie": "Spielman", "Eleisha": "Jackson",
    "Claus": "Wilke"}
```

ANOTHER DATA TYPE: DICTIONARIES

- **Dictionaries are defined with braces: {}**
  - Contain `key:value` pairs
  - Known in other contexts as "associative arrays"

```
# Define a dictionary of names
names = {"Stephanie": "Spielman", "Eleisha": "Jackson",
    "Claus": "Wilke"}

# Each key:value pair is a single item
print len(names)
    3
```

ANOTHER DATA TYPE: DICTIONARIES

- **Dictionaries are defined with braces: {}**
  - Contain `key:value` pairs
  - Known in other contexts as "associative arrays"

```
# Define a dictionary of names
names = {"Stephanie": "Spielman", "Eleisha": "Jackson",
    "Claus": "Wilke"}

# Each key:value pair is a single item
print len(names)
    3

# Index dictionaries via *keys* (not position!!)
print names["Eleisha"]
    "Jackson"
```

ANOTHER DATA TYPE: DICTIONARIES

```
names = {"Stephanie": "Spielman", "Eleisha": "Jackson",
"Claus": "Wilke"}

# Add a key:value pair to a dictionary and print to confirm
names["Bob"] = "Smith"
```

ANOTHER DATA TYPE: DICTIONARIES

```
names = {"Stephanie": "Spielman", "Eleisha": "Jackson",
"Claus": "Wilke"}

# Add a key:value pair to a dictionary and print to confirm
names["Bob"] = "Smith"

print names
    {'Eleisha': 'Jackson', 'Claus': 'Wilke', 'Stephanie':
    'Spielman', 'Bob': 'Smith'}
```

Did you expect this output?

DICTIONARIES ARE UNORDERED

- **Unique key:value pairs are *always* preserved, but their order is not**
  - One of many reasons why we index with keys, not positions

DICTIONARY KEYS MUST BE UNIQUE

- **Keys must be unique, but values may be repeated:**

```
acceptable_dict = {"a": 5, "b": 3, "c": 5}
```

DICTIONARY KEYS MUST BE UNIQUE

- **Keys must be unique, but values may be repeated:**

```
acceptable_dict = {"a": 5, "b": 3, "c": 5}
```

- **Adding an existing key will *overwrite* the original:**

```
acceptable_dict["a"] = 7
print acceptable_dict
    {"a": 7, "c": 5, "b": 3}
```

COMMON DICTIONARY METHODS

```
# Define a dictionary
medals = {"gold": "first", "silver": "second", "bronze":
"third"}


# The .keys() method returns a list of dictionary keys
print medals.keys()
    ['bronze', 'silver', 'gold']


# The .values() method returns a list of dictionary values
print medals.values()
    ['third', 'second', 'first']
```

EXERCISE BREAK

LOGICAL EVALUATIONS AND ITERATIONS

LOGICAL OPERATORS

- **Evaluate a condition as True or False using *logical operators***
  - Variables with True/False values are of a type called *boolean*

Operator	Meaning
==	Equal to
!=	Not equal to
>, <	Greater than; less than
>=, <=	Greater than or equal to ; less than or equal to

PERFORMING LOGICAL COMPARISONS

```
a = 6
b = 120
print b > a
    True

print 6 == 6
    True

print 7 != "this isn't even a number"
    True

c = 1.2345
print -99 >= c
    False
```

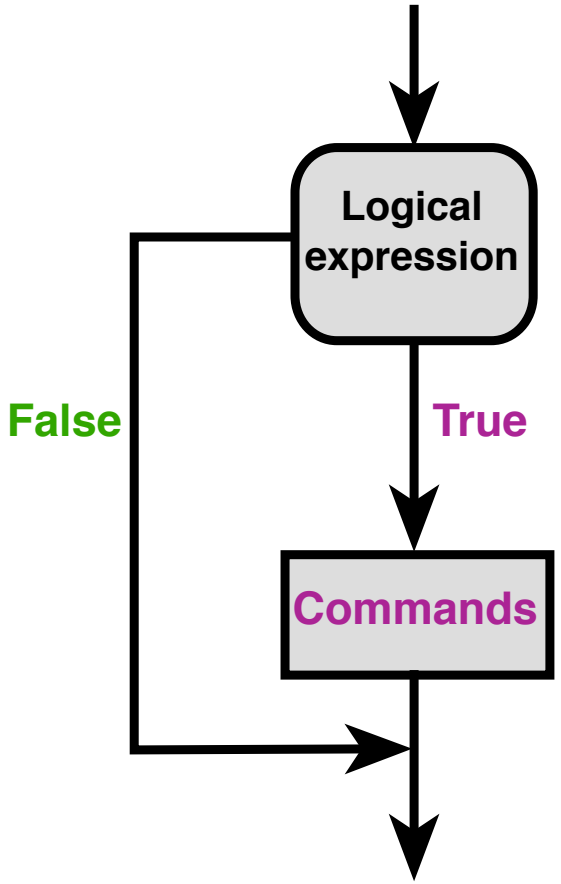
COMBINING LOGICAL STATEMENTS

- **Use the Python operators `and` and `or` to combine logical statements**
  - `and`: *both* conditions must be True
  - `or`: *only one* condition must be True

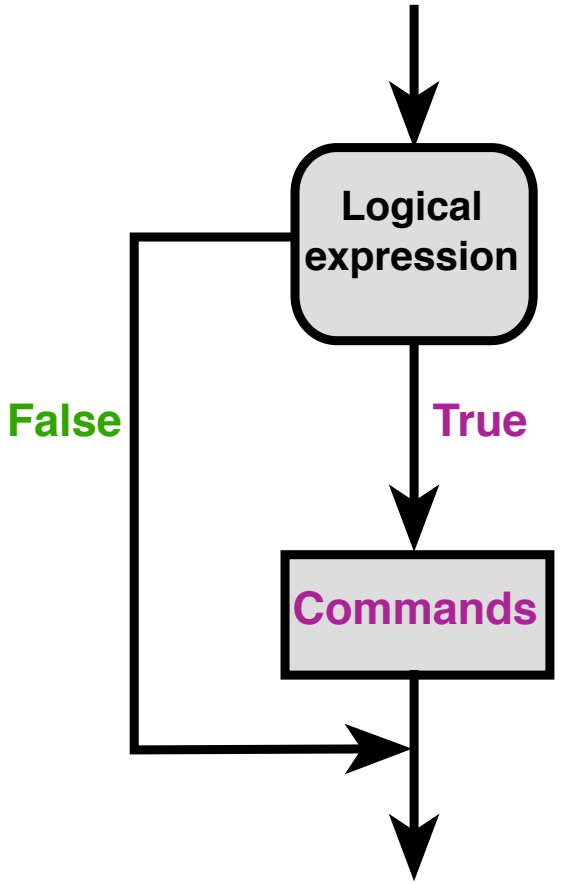
COMBINING LOGICAL STATEMENTS

```
a = 6
b = 120
c = ["z", "x", "y", "w"]

print (a == 6 and b == 120)
    True

print (a == 6 or b == 92)
    True

print (b < 10 or a > 55)
    False

print (b != 7 and len(c) <= 8)
    True

print (c[1] == "x" and a == b)
    False
```

PROGRAM CONTROL FLOW WITH IF STATEMENTS

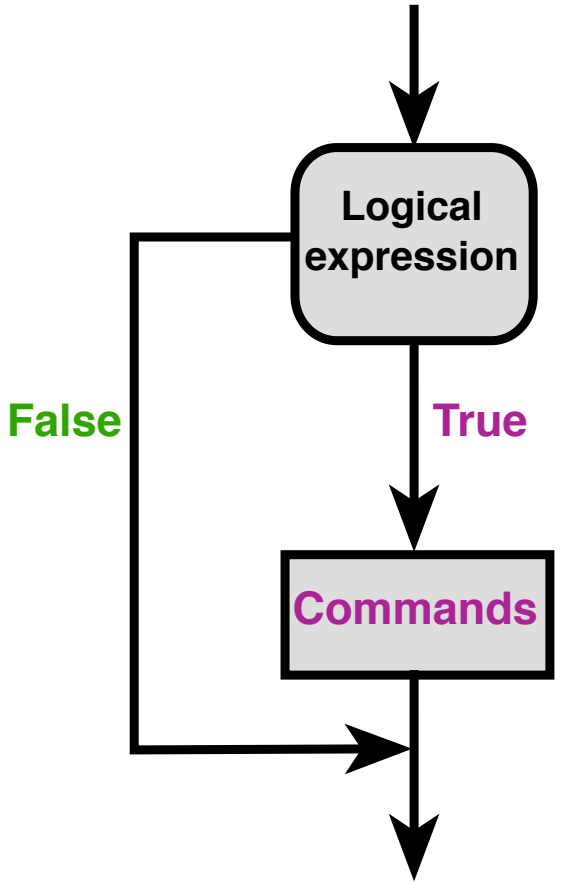


PROGRAM CONTROL FLOW WITH IF STATEMENTS



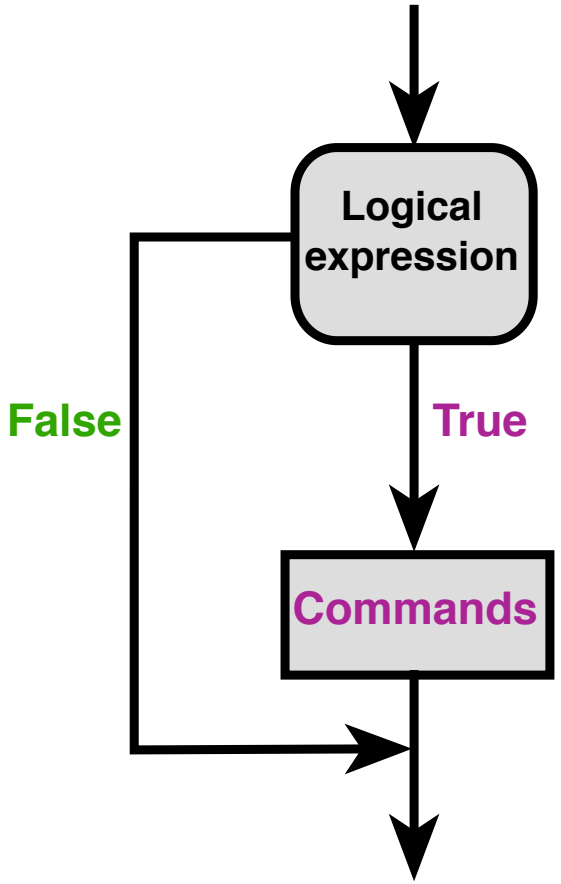

```
...
...      Python code
...

if logical expression :
    ...
    ...      Code run if True
    ...

...
...      Python code
...
```

PROGRAM CONTROL FLOW WITH IF STATEMENTS



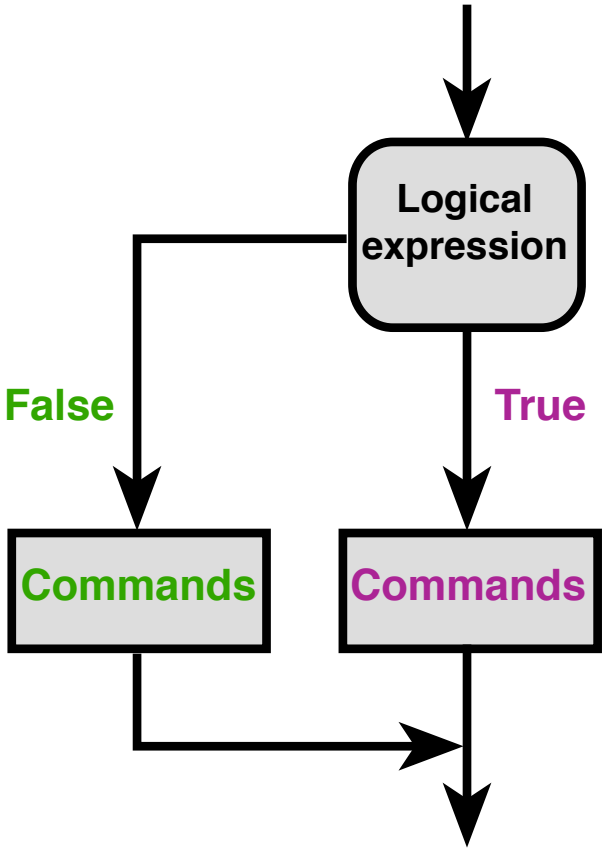

```
...
...      Python code
...

if logical expression :
    ...
    ...      Code run if True
    ...

...
...      Python code
...
```

PROGRAM CONTROL FLOW WITH IF STATEMENTS



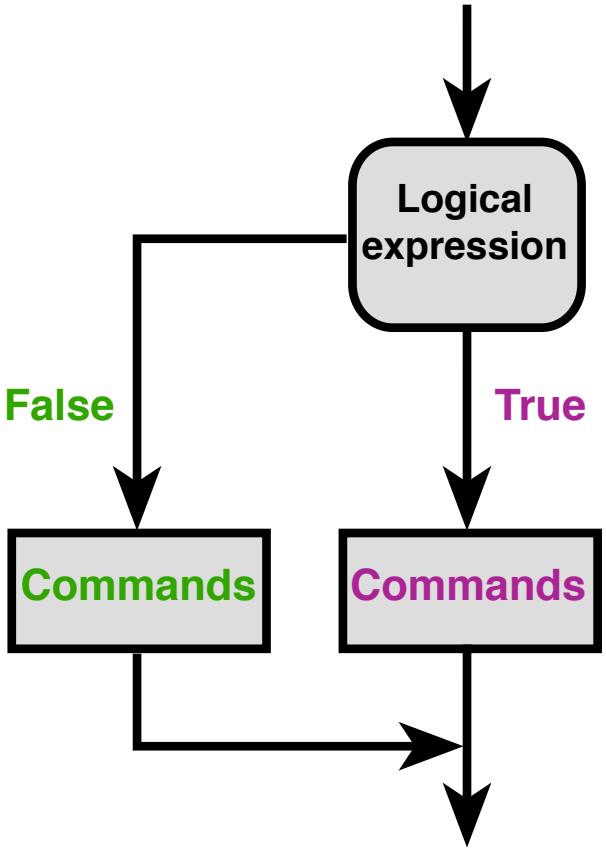

```
a = 7
b = 5
if a > b:
    print "a is bigger"

    a is bigger
```

IF-ELSE STATEMENTS



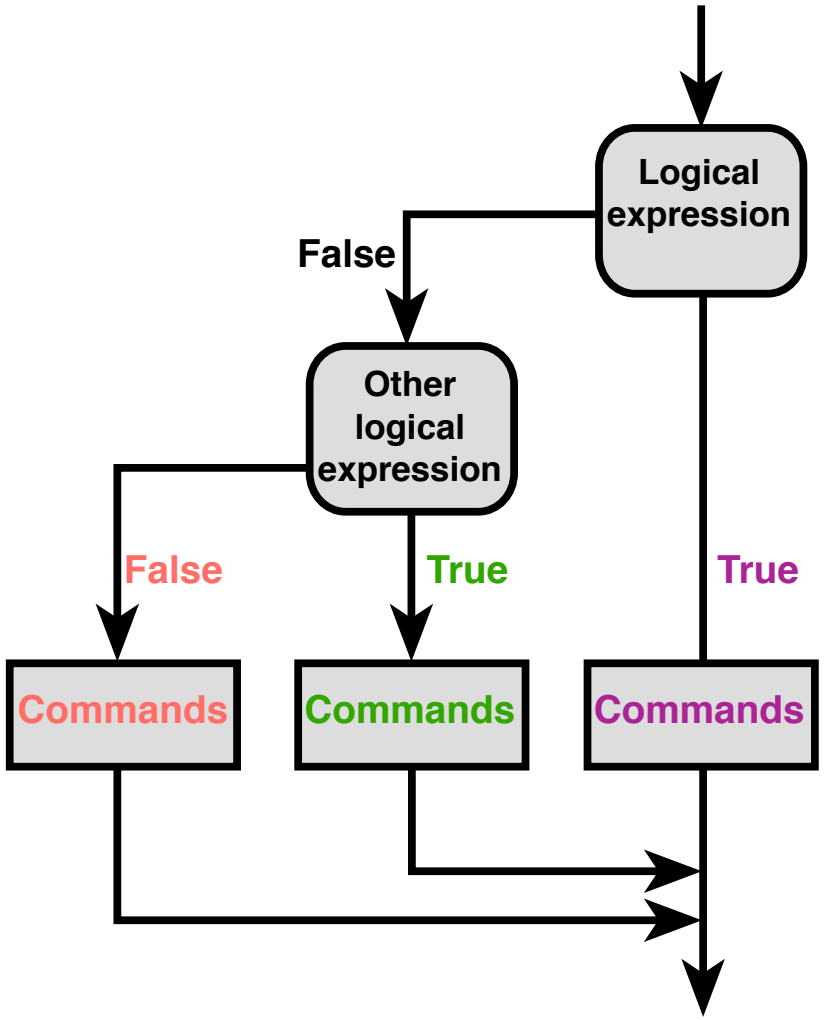

```
...
...   Python code
...

if logical expression :
    ...
    ...   Code run if True
    ...
else:
    ...
    ...   Code run if False
    ...

...
...   Python code
...
```

IF-ELSE STATEMENTS



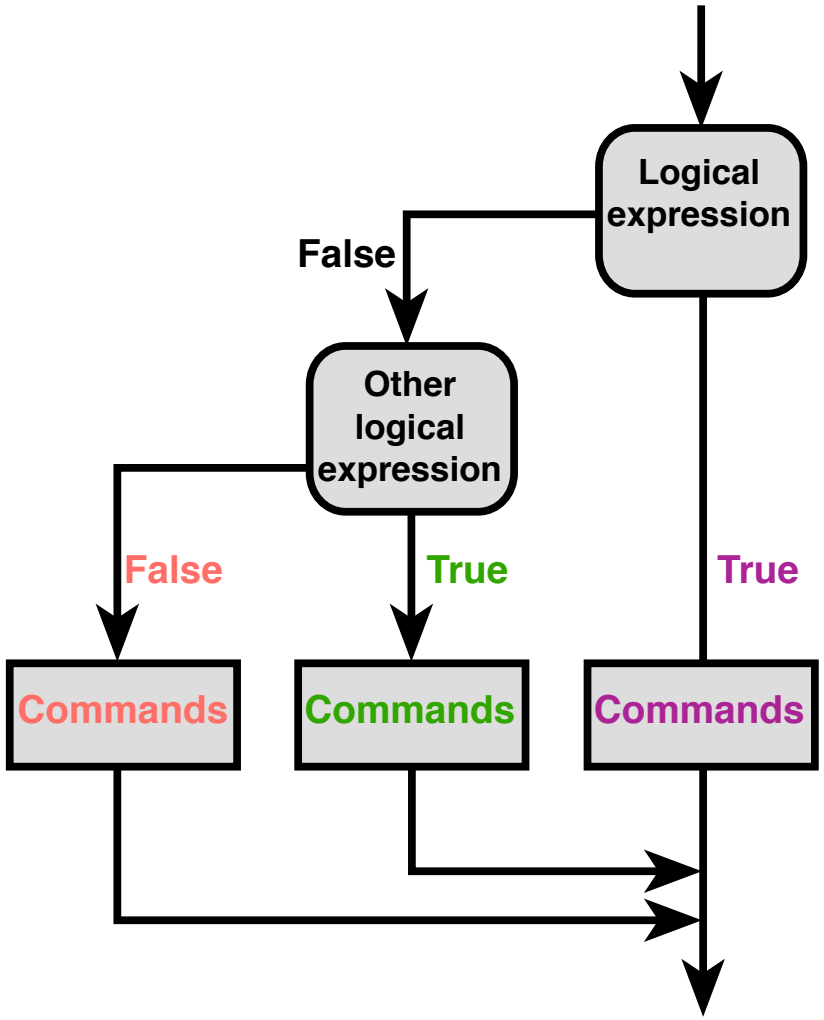

```
a = 7
b = 5
if a > b:
    print "a is bigger"
else:
    print "a is not bigger"

a is bigger
```

IF-ELIF-ELSE STATEMENTS



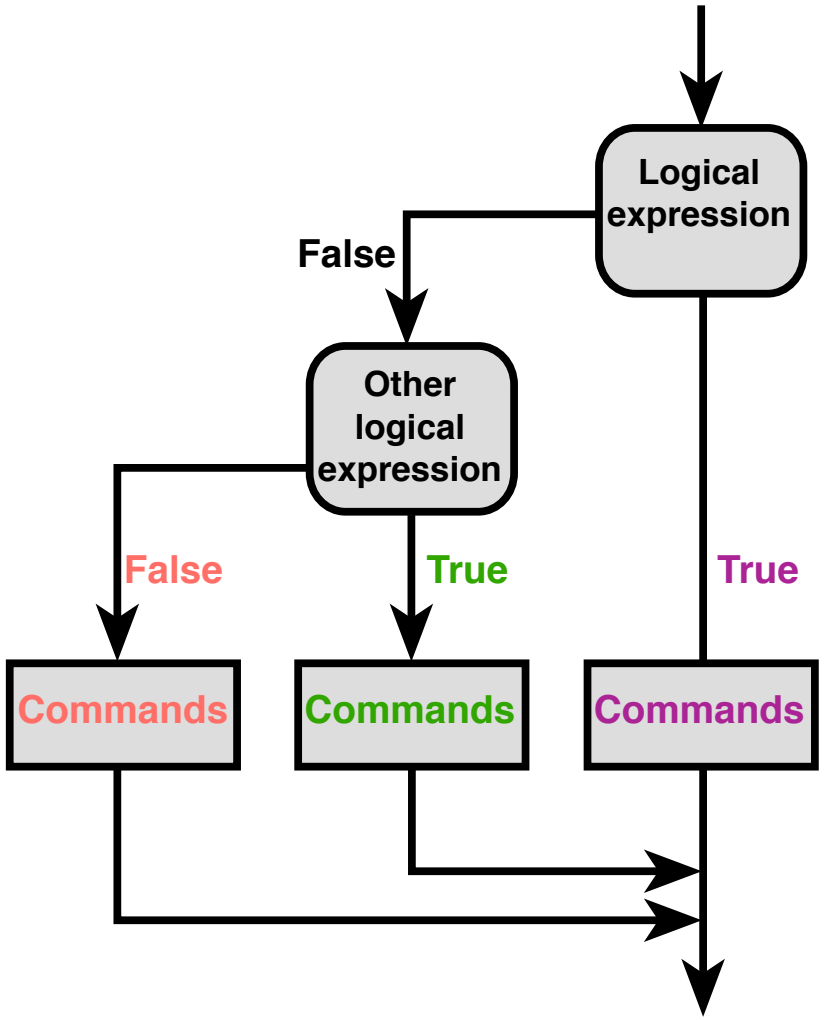

```
...
...   | Python code
...

if logical expression :
    ...
    ...   | Code run if True
    ...
elif other logical expr :
    ...
    ...   | Code run if True
    ...
else:
    ...
    ...   | Code run if *all* False
    ...

...
...   | Python code
...
```

IF-ELIF-ELSE STATEMENTS




```
a = 7
b = 5
if a > b:
    print "a is bigger"
elif a < b:
    print "b is bigger"
else:
    print "a is equal to b"
```

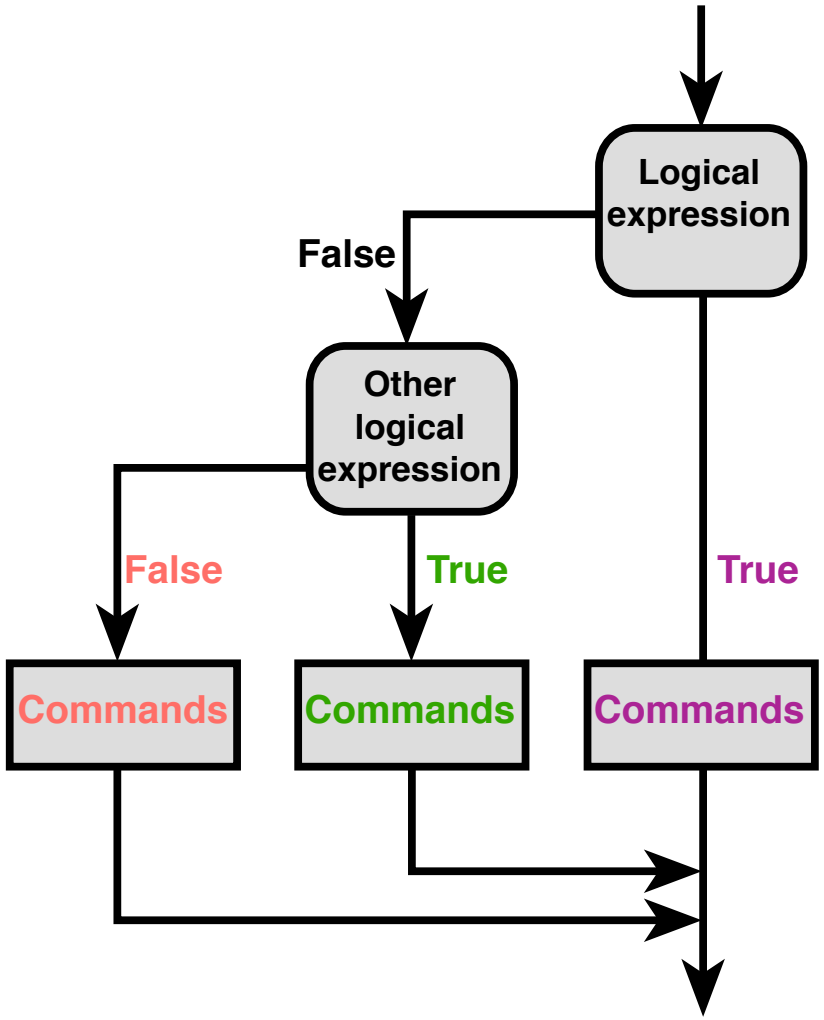
IF-ELIF-ELSE STATEMENTS




```
a = 7
b = 5
if a > b:
    print "a is bigger"
elif a < b:
    print "b is bigger"
else:
    print "a is equal to b"
```

You can have as many elif statements as you want!

IF-ELIF-ELSE STATEMENTS




```
# You don't need to end with else
a = 7
b = 5
if a > b:
    print "a is bigger"
elif a < b:
    print "b is bigger"
elif a == b:
    print "a is equal to b"
```

EXERCISE BREAK

ITERATION IS OUR OTHER CONTROL FLOW TOOL

- **Iteration performs the same code repeatedly**

ITERATION IS OUR OTHER CONTROL FLOW TOOL

- **Iteration performs the same code repeatedly**

- **Two flavors:**
  - For-loops iterate a pre-specified number of times
  - While-loops iterate while a logical condition remains True

ITERATION IS OUR OTHER CONTROL FLOW TOOL

- **Iteration performs the same code repeatedly**

- **Two flavors:**
  - For-loops iterate a pre-specified number of times
  - While-loops iterate while a logical condition remains True

ITERATING WITH FOR-LOOPS

- **Two basic uses:**
  - Perform a task on each item in a list, dictionary, etc.
  - Perform a task a certain number of times

ITERATING OVER LISTS

```
# Generic list
some_list = [...items...]

# Loop over the list
for item in some_list:
    ...
    ...  | Do these commands for each item
    ...

...
...  | Back to the rest of the script
...
```

ITERATING OVER LISTS

```
# Generic list
some_list = [...items...]

# Loop over the list
for item in some_list:
    ...
    ...    Do these commands for each item
    ...

...
...    Back to the rest of the script
...
```

ITERATING OVER LISTS

```
# Generic list
some_list = [...items...]

# Loop over the list
for item in some_list:
    ...
    ...    Do these commands for each item
    ...

...
...    Back to the rest of the script
...
```

ITERATING OVER LISTS

```
# Generic list
some_list = [...items...]

# Loop over the list
for item in some_list:
    ...
    ...     Do these commands for each item
    ...

...
...     Back to the rest of the script
...
```

`item` is the loop variable.

- Takes on a new value for each iteration
- The name is *arbitrary*

ITERATING OVER LISTS

```
# Generic list
some_list = [...items...]

# Loop over the list
for blahblahblah in some_list:
    ...
    ...  | Do these commands for each item
    ...

...
...  | Back to the rest of the script
...
```

`item` is the loop variable.

- Takes on a new value for each iteration
- The name is *arbitrary*

ITERATING OVER LISTS

```
# Generic list
some_list = [...items...]

# Loop over the list
for x in some_list:
    ...
    ...  | Do these commands for each item
    ...

...
...  | Back to the rest of the script
...
```

`item` is the loop variable.

- Takes on a new value for each iteration
- The name is *arbitrary*

ITERATING OVER LISTS EXAMPLE: CURVING GRADES

```
# List of grades
grades = [88, 71, 74, 83, 57, 79, 66]

# Loop over the grades
for grade in grades:
    print grade * 1.1
```

```
96.8
78.1
81.4
91.3
62.7
86.9
72.6
```

ITERATING OVER LISTS EXAMPLE: CURVING GRADES

```
# List of grades
grades = [88, 71, 74, 83, 57, 79, 66]

# Empty list of new grades to populate
new_grades = []

# Loop over the grades and save curved grade
for grade in grades:
    new = grade * 1.1
    new_grades.append(new)

print new_grades
    [96.8, 78.1, 81.4, 91.3, 62.7, 86.9, 72.6]
```

BUT NO HARD-CODING!!!

```
# List of grades
grades = [88, 71, 74, 83, 57, 79, 66]

# Empty list of new grades to populate
new_grades = []

# Loop over the grades and save curved grade
for grade in grades:
    new = grade * 1.1
    new_grades.append(new)

print new_grades
    [96.8, 78.1, 81.4, 91.3, 62.7, 86.9, 72.6]
```

BUT NO HARD-CODING!!!

```
# List of grades
grades = [88, 71, 74, 83, 57, 79, 66]

# Empty list of new grades to populate
new_grades = []

# Curving value
curve = 1.1

# Loop over the grades and save curved grade
for grade in grades:
    new = grade * curve
    new_grades.append(new)

print new_grades
    [96.8, 78.1, 81.4, 91.3, 62.7, 86.9, 72.6]
```

USE A COUNTER VARIABLE TO KEEP TRACK OF LOOP

```
for grade in grades:
    print grade * 1.1
```

```
96.8
78.1
81.4
91.3
62.7
86.9
72.6
```

```
i = 0
for grade in grades:
    print "Iteration " + str(i)
    print grade * 1.1
    i += 1
```

```
Iteration 0
96.8
Iteration 1
78.1
Iteration 2
81.4
Iteration 3
91.3
Iteration 4
62.7
...
```

ITERATING A CERTAIN NUMBER OF TIMES

- **Use the `range()` function**
  - This function defines a list, using the same arguments as *indexing*

```
print range(10)
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]


print range(5, 12)
    [5, 6, 7, 8, 9, 10, 11]
```

ITERATING A CERTAIN NUMBER OF TIMES

- **Use the `range()` function**
  - This function defines a list, using the same arguments as *indexing*

```
print range(10)
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

This list has a length of 10!

```
print range(5, 12)
    [5, 6, 7, 8, 9, 10, 11]
```

ITERATING A CERTAIN NUMBER OF TIMES

```
# List of grades
grades = [88, 71, 74, 83, 57, 79, 66]
```

```
for grade in grades:
    print grade * 1.1
```

```
96.8
78.1
81.4
91.3
62.7
86.9
72.6
```

```
for i in range(len(grades)):
    print grades[i] * 1.1
```

```
96.8
78.1
81.4
91.3
62.7
86.9
72.6
```

LOOPING OVER STRINGS

```
for s in "python":
    print s
```

```
p
y
t
h
o
n
```

LOOPING OVER DICTIONARIES

```
price = {"banana" : 0.79, "apple": 1.02, "bell pepper": 2.39}
for item in price:
    print item
```

LOOPING OVER DICTIONARIES

```
price = {"banana" : 0.79, "apple": 1.02, "bell pepper": 2.39}
for item in price:
    print item
```

```
bell pepper
banana
apple
```

What are we actually looping over?

LOOPING OVER DICTIONARIES

```
price = {"banana" : 0.79, "apple": 1.02, "bell pepper": 2.39}
for item in price:
    # Print the key *and* value
    print item, price[item]
```

```
bell pepper, 2.39
banana, 0.79
apple, 1.02
```

USING IF AND FOR TOGETHER

```
# List of grades
grades = [88, 71, 74, 83, 57, 79, 66]

# Empty list of letter
letter_grades = []

# Determine the letter grade
for grade in grades:
    if grade >= 90:
        letter_grades.append("A")
    elif grade >= 80:
        letter_grades.append("B")
    elif grade >= 70:
        letter_grades.append("C")
    elif grade >= 60:
        letter_grades.append("D")
    else:
        letter_grades.append("F")

print letter_grades
    ['B', 'C', 'C', 'B', 'F', 'C', 'D']
```

CONTROLLING THE LOOPS EVEN MORE

- **Two statements change loop flow:**
  - `continue`
    - immediately start the next iteration and skip remaining loop statements
  - `break`
    - immediately exit out of loop entirely

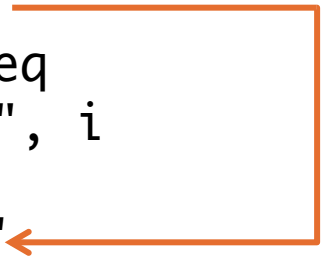
THE CONTINUE STATEMENT

```
codons = ["ATT", "GAT", "NNA", "ANG", "NTT", "ATG"]

# Print unambiguous codons only
i = 0
for seq in codons:
    i += 1
    if "N" in seq:
        continue     # Immediately start next iteration
    print "The sequence is " + seq
    print "loop iteration count:", i
```

```
The sequence is ATT
loop iteration count: 1
The sequence is GAT
loop iteration count: 2
The sequence is ATG
loop iteration count: 6
```

THE BREAK STATEMENT

```
codons = ["ATT", "GAT", "NNA", "ANG", "NTT", "ATG"]

# Print unambiguous codons only
i = 0
for seq in codons:
    i += 1
    if "N" in seq:
        print "Oh no, ambiguities! I'm gonna stop."
        break  # Immediately exit
    print "The sequence is " + seq
    print "loop iteration count:", i

print "Outside of the loop now."
```



```
The sequence is ATT
loop iteration count: 1
The sequence is GAT
loop iteration count: 2
Outside of the loop no.
```

THE BREAK STATEMENT

```
codons = ["ATT", "GAT", "NNA", "ANG", "NTT", "ATG"]

# Print unambiguous codons only
i = 0
for seq in codons:
    i += 1
    if "N" in seq:
        print "Oh no, ambiguities! I'm gonna stop."
        break  # Immediately exit
    print "The sequence is " + seq
    print "loop iteration count:", i

print "Outside of the loop now."
```

```
The sequence is ATT
loop iteration count: 1
The sequence is GAT
loop iteration count: 2
Outside of the loop no.
```

NB: these are essentially required for while-loops

EXERCISE BREAK