

Lecture 6

Control Structures

Conditionals: If-Statements

Format

```
if <boolean-expression>:  
    <statement>  
    ...  
    <statement>
```

Example

```
# Put x in z if it is positive  
if x > 0:  
    z = x
```

Execution:

if <boolean-expression> is true, then execute all of the statements indented directly underneath (until first non-indented statement)

Conditionals: If-Else-Statements

Format

```
if <boolean-expression>:  
    <statement>  
    ...  
else:  
    <statement>  
    ...
```

Example

```
# Put max of x, y in z  
if x > y:  
    z = x  
else:  
    z = y
```

Execution:

if <boolean-expression> is true, then execute statements indented under if; otherwise execute the statements indented under elsec

Conditionals: If-Elif-Else-Statements

Format

```
if <boolean-expression>:  
    <statement>  
    ...  
elif <boolean-expression>:  
    <statement>  
    ...  
...  
else:  
    <statement>  
    ...
```

Example

```
# Put max of x, y, z in w  
if x > y and x > z:  
    w = x  
elif y > z:  
    w = y  
else:  
    w = z
```

Conditionals: If-Elif-Else-Statements

Format

```
if <boolean-expression>:  
    <statement>  
    ...  
elif <boolean-expression>:  
    <statement>  
    ...  
...  
else:  
    <statement>  
    ...
```

Notes on Use

- No limit on number of elif
 - Can have as many as want
 - Must be between if, else
- The else is always optional
 - if-elif by itself is fine
- Booleans checked in order
 - Once it finds a true one, it skips over all the others
 - else means **all** are false

Conditional Expressions

Format

`e1 if bexp else e2`

- `e1` and `e2` are any expression
- `bexp` is a boolean expression
- This is an expression!

Example

`# Put max of x, y in z`

`z = x if x > y else y`



expression,
not statement

Program Flow vs. Local Variables

```
def max(x,y):
```

```
    """Returns: max of x, y"""
```

```
    # swap x, y
```

```
    # put the larger in y
```

```
1  if x > y:
```

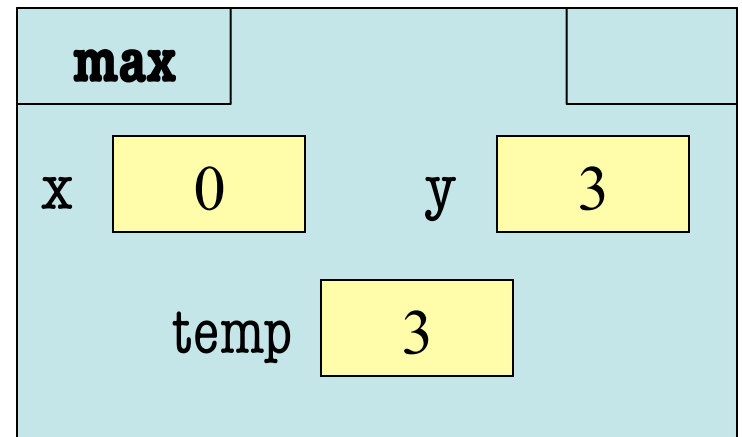
```
2      temp = x
```

```
3      x = y
```

```
4      y = temp
```

```
5  return y
```

- temp is needed for swap
 - x = y loses value of x
 - “Scratch computation”
 - Primary role of local vars
- max(3,0):



Program Flow vs. Local Variables

```
def max(x,y):
```

```
    """Returns: max of x, y"""
```

```
    # swap x, y
```

```
    # put the larger in y
```

```
    if x > y:
```

```
        temp = x
```

```
        x = y
```

```
        y = temp
```

```
    return temp
```

- Value of max(3,0)?

A: 3

B: 0

C: **Error!**

D: I do not know

Program Flow vs. Local Variables

```
def max(x,y):
```

```
    """Returns: max of x, y"""
```

```
    # swap x, y
```

```
    # put the larger in y
```

```
    if x > y:
```

```
        temp = x
```

```
        x = y
```

```
        y = temp
```

```
    return temp
```

- Value of max(3,0)?

A: 3 CORRECT

B: 0

C: Error!

D: I do not know

- Local variables last until
 - They are deleted or
 - End of the function
- Even if defined inside **if**

Program Flow vs. Local Variables

def max(x,y):

"""Returns: max of x, y"""

swap x, y

put the larger in y

if x > y:

temp = x

x = y

y = temp

return temp

- Value of max(0,3)?

A: 3

B: 0

C: **Error!**

D: I do not know

Program Flow vs. Local Variables

```
def max(x,y):
```

```
    """Returns: max of x, y"""
```

```
    # swap x, y
```

```
    # put the larger in y
```

```
    if x > y:
```

```
        temp = x
```

```
        x = y
```

```
        y = temp
```

```
    return temp
```

- Value of max(0,3)?

A: 3

B: 0

C: **Error!** **CORRECT**

D: I do not know

- Variable existence depends on flow
- Understanding flow is important in testing

Local Variables Revisited

- Never refer to a variable that might not exist
- Variable “scope”
 - Block (indented group) where it was first assigned
 - Way to think of variables; not actually part of Python
- **Rule of Thumb:** Limit variable usage to its scope

```
def max(x,y):
```

```
    """Returns: max of x, y"""
```

```
    # swap x, y
```

```
    # put larger in temp
```

```
    if x > y:
```

```
        temp = x
```

```
        x = y
```

```
        y = temp
```

```
    return temp
```

First assigned

Outside scope

Local Variables Revisited

- Never refer to a variable that might not exist
- Variable “scope”
 - Block (indented group) where it was first assigned
 - Way to think of variables; not actually part of Python
- **Rule of Thumb:** Limit variable usage to its scope

```
def max(x,y):
```

```
    """Returns: max of x, y"""
```

```
    # swap x, y
```

```
    # put larger in temp
```

```
    temp = y
```

```
    if x > y:
```

```
        | temp = x
```

```
    return temp
```

First assigned

Inside scope

Variation on max

```
def max(x,y):
```

```
    """Returns:  
    max of x, y"""
```

```
    if x > y:
```

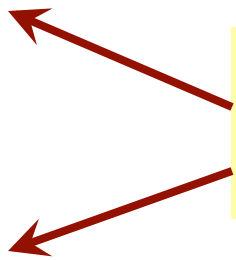
```
        return x
```

```
    else:
```

```
        return y
```

Which is better?
Matter of preference

There are two **returns**!
But only one is executed



Beyond Sequences: The while-loop

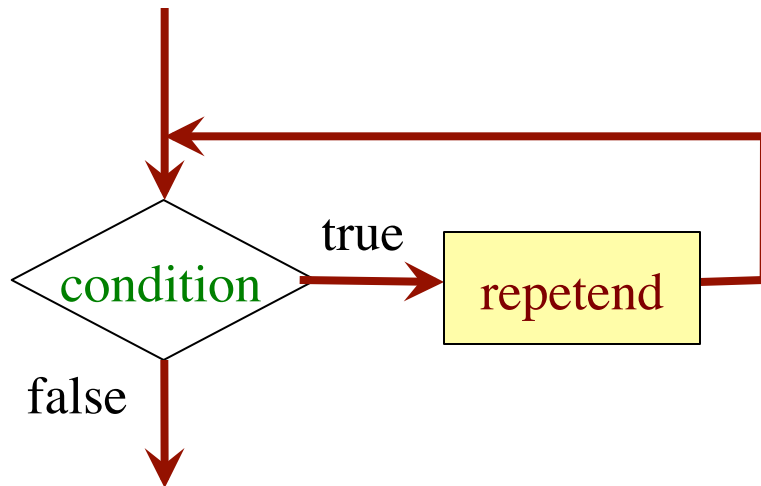
while *<condition>*:

statement 1

...

statement n

repetend or **body**



- Relationship to for-loop
 - Broader notion of “still stuff to do”
 - Must explicitly ensure condition becomes false

while Versus for

```
# process range b..c
for k in range(b,c+1)
    process k
```

Must remember to increment

```
# process range b..c
k = b
while k <= c:
    process k
    k = k+1
```

- Makes list $c+1-b$ elements
- List uses up memory
- Impractical for large ranges

- Just needs an int
- Much less memory usage
- Best for large ranges

For Loops: Processing Sequences

```
# Print contents of seq
x = seq[0]
print x
x = seq[1]
print x
...
x = seq[len(seq)-1]
print x
```

- **Remember:**
 - Cannot program ...
 - Reason for recursion

The for-loop:

```
for x in seq:
    print x
```

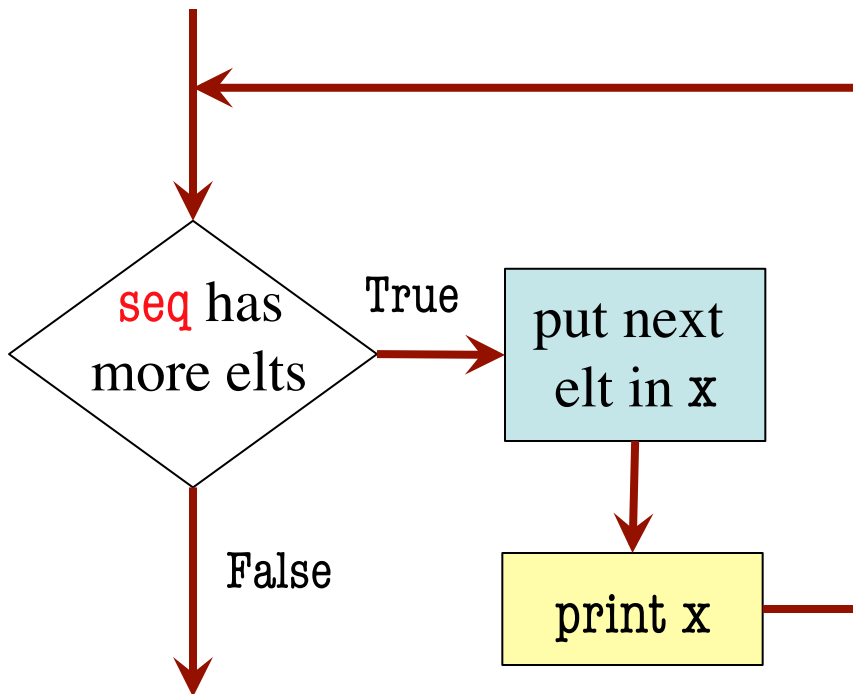
- Key Concepts
 - **loop sequence:** seq
 - **loop variable:** x
 - **body:** print x
 - Also called **repetend**

For Loops: Processing Sequences

The for-loop:

```
for x in seq:  
    print x
```

- loop sequence: **seq**
- loop variable: **x**
- body: **print x**



To execute the for-loop:

1. Check if there is a “next” element of **loop sequence**
2. If not, terminate execution
3. Otherwise, put the element in the **loop variable**
4. Execute all of **the body**
5. Repeat as long as 1 is true

More Complex For-Loops

- Combine with a *counter*
 - Variable that increments each time body executed
 - Tracks position in seq
- Nest conditionals inside
 - Body is all indented code
 - Can put other control structures inside the body

- **Example:**

```
cnt = 0
```

```
for x in seq:
```

```
    print `x` +' at '+' `cnt`
```

```
    cnt = cnt + 1 # incr
```

- **Example:**

```
nints = 0 # num of ints
```

```
for x in seq:
```

```
    if type(x) == int:
```

```
        nints = nints + 1
```

while Versus for

incr seq elements

for *k* in range(len(seq)):

seq[k] = seq[k]+1

Makes a **second** list.

incr seq elements

k = 0

while *k* < len(seq):

seq[k] = seq[k]+1

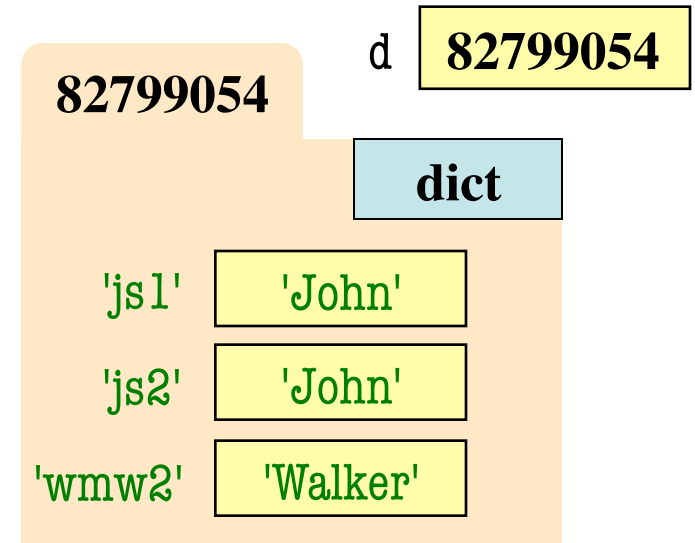
k = *k*+1

while is more flexible, but
is **much trickier** to use

Using Dictionaries (Type dict)

- Access elts. like a list
 - `d['js1']` evaluates to `'John'`
 - But cannot slice ranges!
- Dictionaries are **mutable**
 - Can reassign values
 - `d['js1'] = 'Jane'`
 - Can add new keys
 - `d['aa1'] = 'Allen'`
 - Can delete keys
 - `del d['wmw2']`

```
d = {'js1':'John','js2':'John',  
      'wmw2':'Walker'}
```

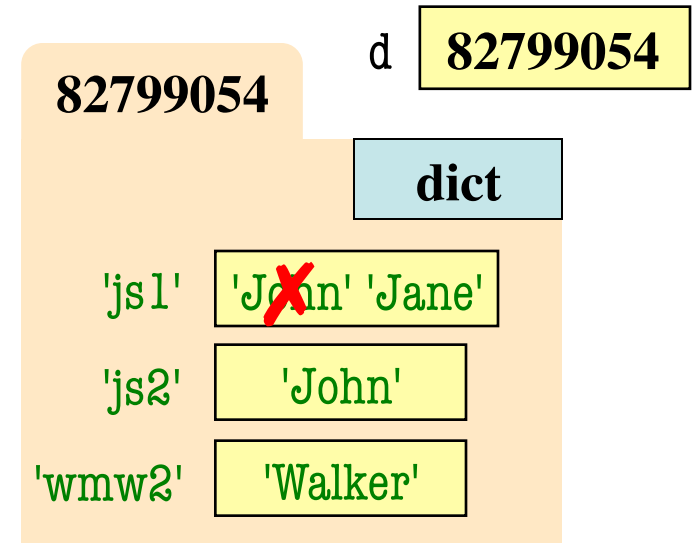


Key-Value order in folder is not important

Using Dictionaries (Type dict)

- Access elts. like a list
 - `d['js1']` evaluates to `'John'`
 - But cannot slice ranges!
- Dictionaries are **mutable**
 - Can reassign values
 - `d['js1'] = 'Jane'`
 - Can add new keys
 - `d['aa1'] = 'Allen'`
 - Can delete keys
 - `del d['wmw2']`

```
d = {'js1':'John','js2':'John',  
      'wmw2':'Walker'}
```

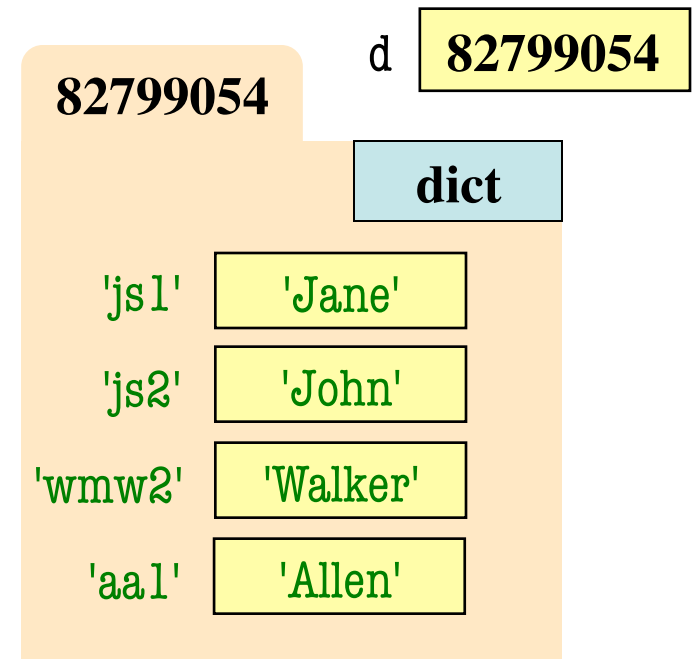


Key-Value order in folder is not important

Using Dictionaries (Type dict)

- Access elts. like a list
 - `d['js1']` evaluates to `'John'`
 - But cannot slice ranges!
- Dictionaries are **mutable**
 - Can reassign values
 - `d['js1'] = 'Jane'`
 - Can add new keys
 - `d['aa1'] = 'Allen'`
 - Can delete keys
 - `del d['wmw2']`

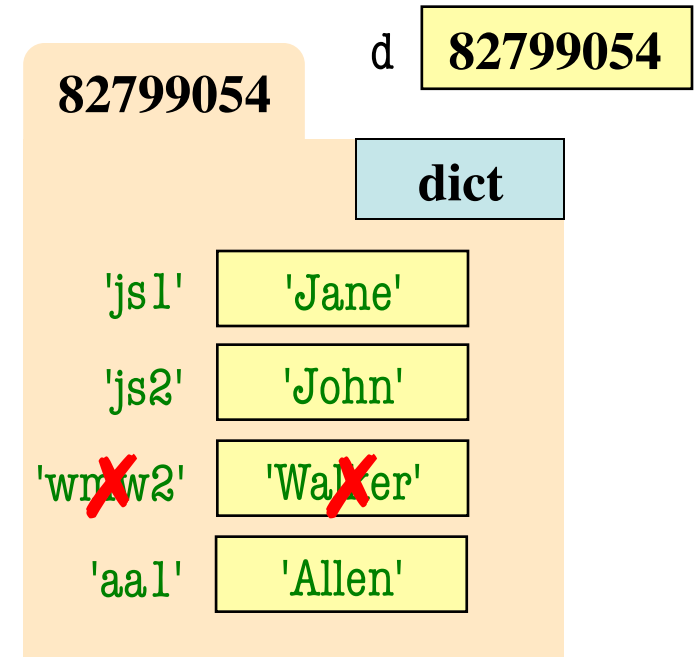
```
d = {'js1':'John','js2':'John',  
      'wmw2':'Walker'}
```



Using Dictionaries (Type dict)

- Access elts. like a list
 - `d['js1']` evaluates to `'John'`
 - But cannot slice ranges!
- Dictionaries are **mutable**
 - Can reassign values
 - `d['js1'] = 'Jane'`
 - Can add new keys
 - `d['aa1'] = 'Allen'`
 - Can delete keys
 - `del d['wmw2']`

```
d = {'js1':'John','js2':'John',  
      'wmw2':'Walker'}
```



Deleting key deletes both