Intro to Computer Science

Previous

- for-loop review (refer to previous readings)
- Control statements
  - if-else

Readings	
Gaddis	• Chapter 3

Next

- Control statements
  - else if
  - nested structures
- while-loops

Readings	
Gaddis	• Chapter 3.4 • Chapter 4.2

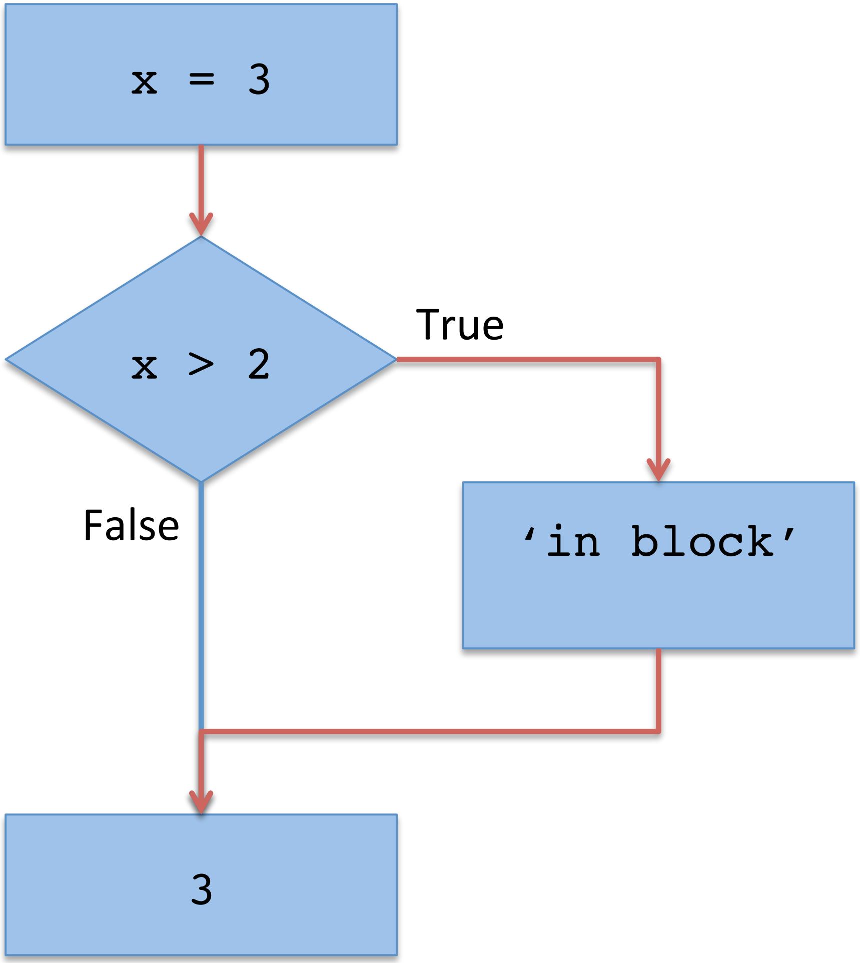
Conditional statements

- Decision structures are implemented using *conditional statements*
- Realized in programming languages through *if-statements*

```
if expression:
  block
else:
  block
```

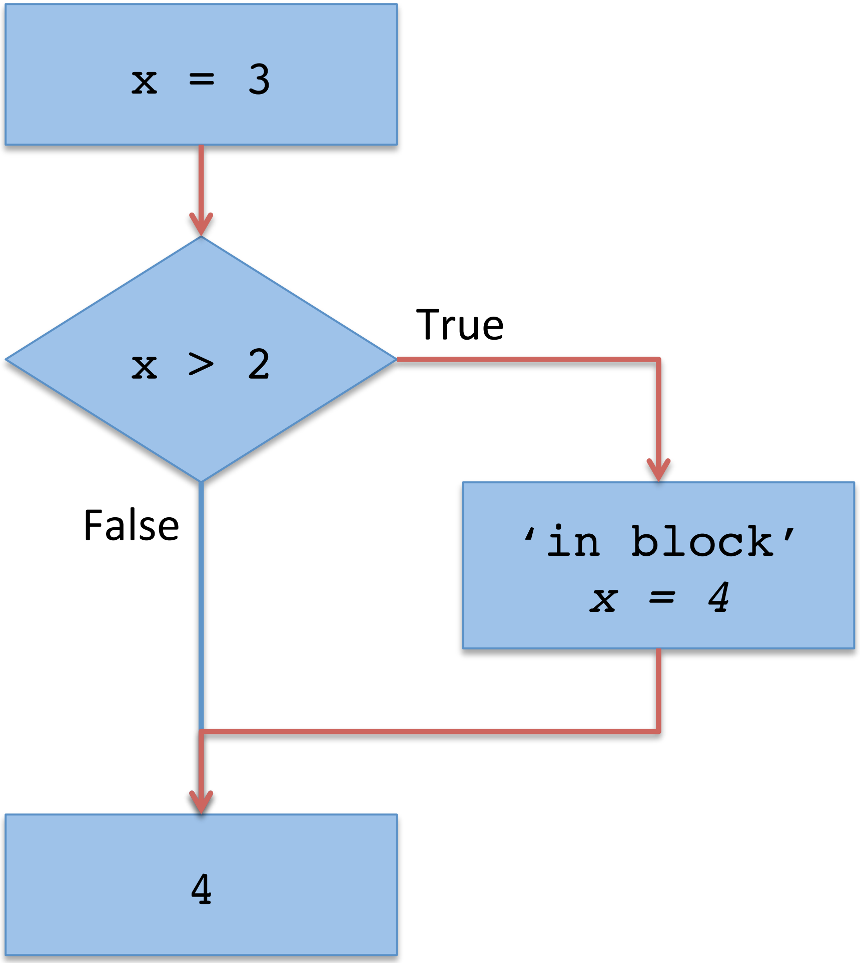
What’s going on here?

```
x = 3
if x > 2:
   print(‘in block’)
print(x)
```



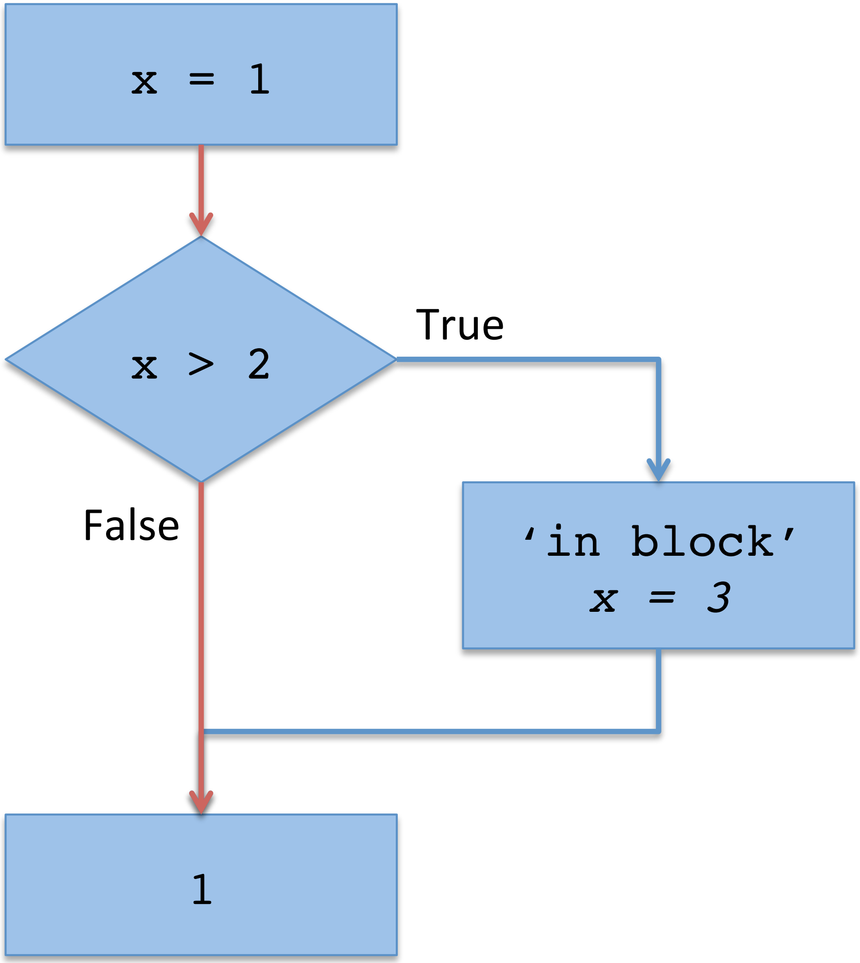
Remember: what happens in the block

```
x = 3
if x > 2:
   print('in block')
   x = 4
print(x)
```



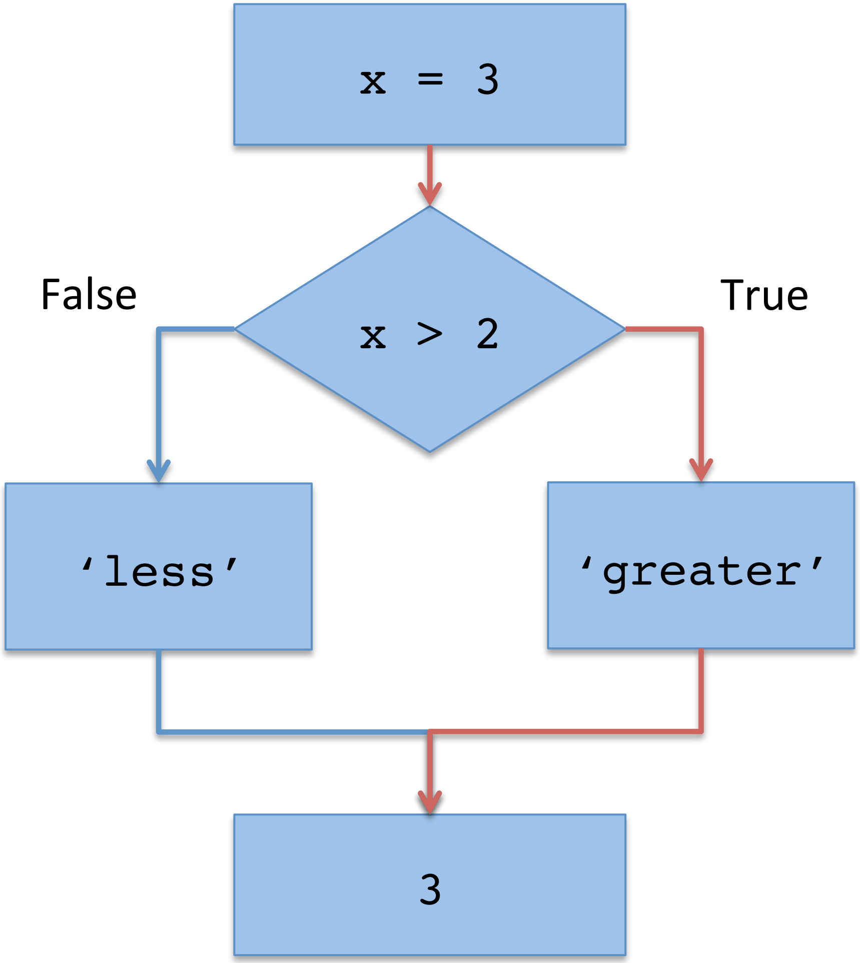
Just passing through

```
x = 1
if x > 2:
   print('in block')
   x = 3
print(x)
```



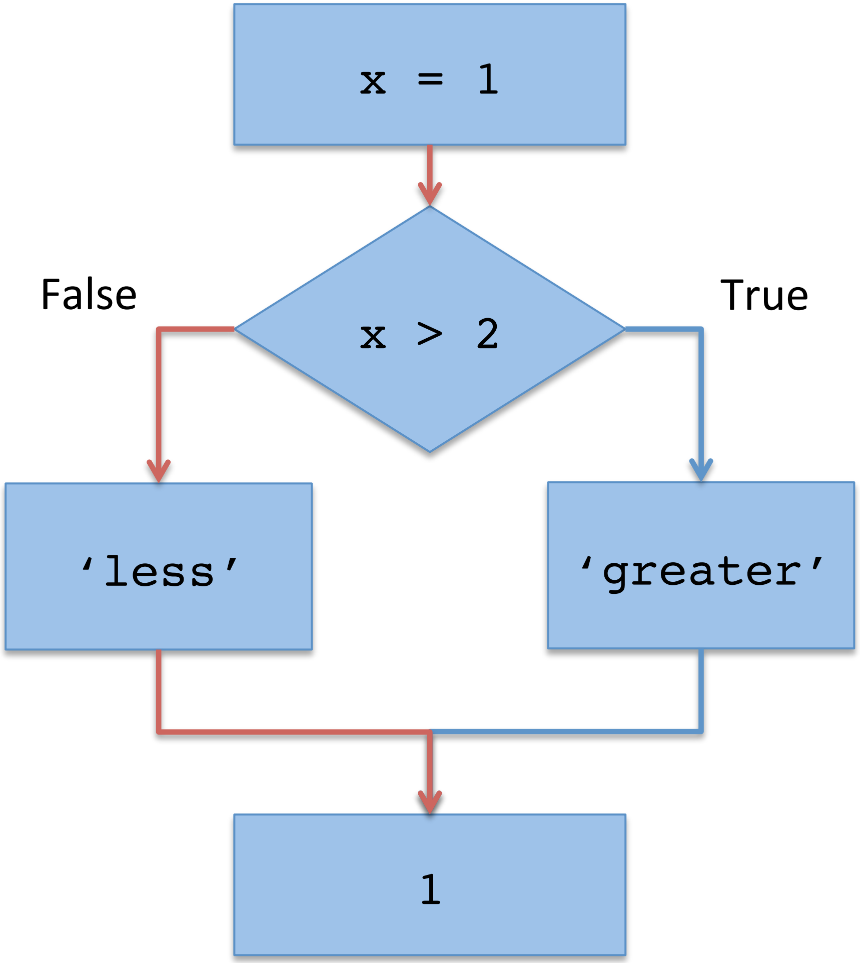
So what’s going on this time?

```
x = 3
if x > 2:
   print(‘greater’)
else:
   print(‘less’)
print(x)
```



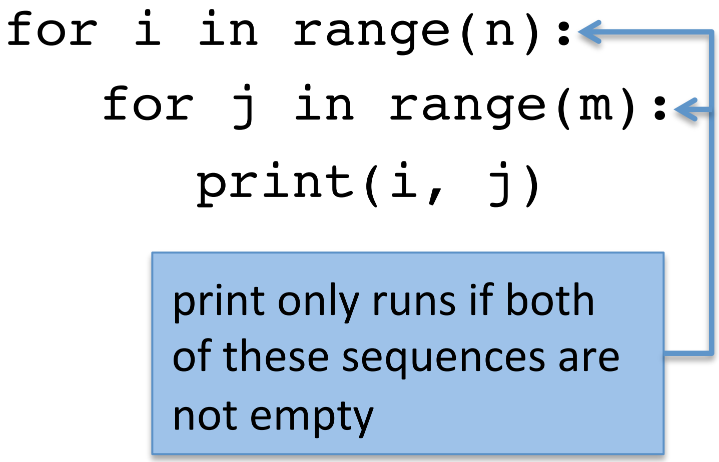
So what's going on this time?

```
x = 1
if x > 2:
   print('greater')
else:
   print('less')
print(x)
```



Block nesting

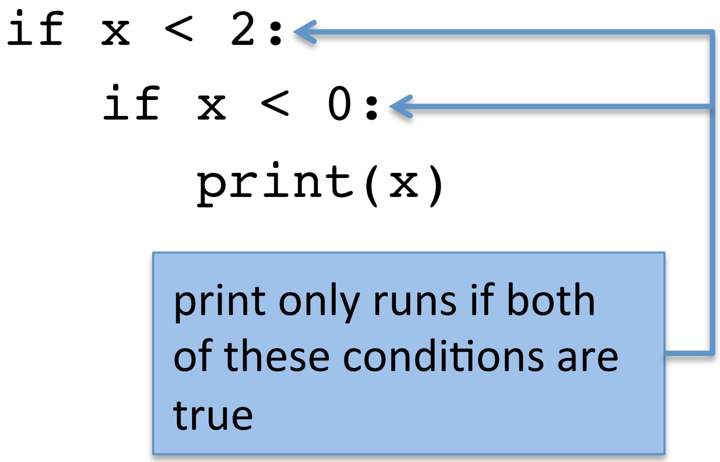
- We saw this with for-loops



```
for i in range(n):
    for j in range(m):
        print(i, j)
```

print only runs if both of these sequences are not empty

- We see it again with if-statements



```
if x < 2:
    if x < 0:
        print(x)
```

print only runs if both of these conditions are true

But for-loops don’t have *else*

- Once a for-loop is finished, the block that follows is always run

- Once an if condition is finished, the block that runs depends

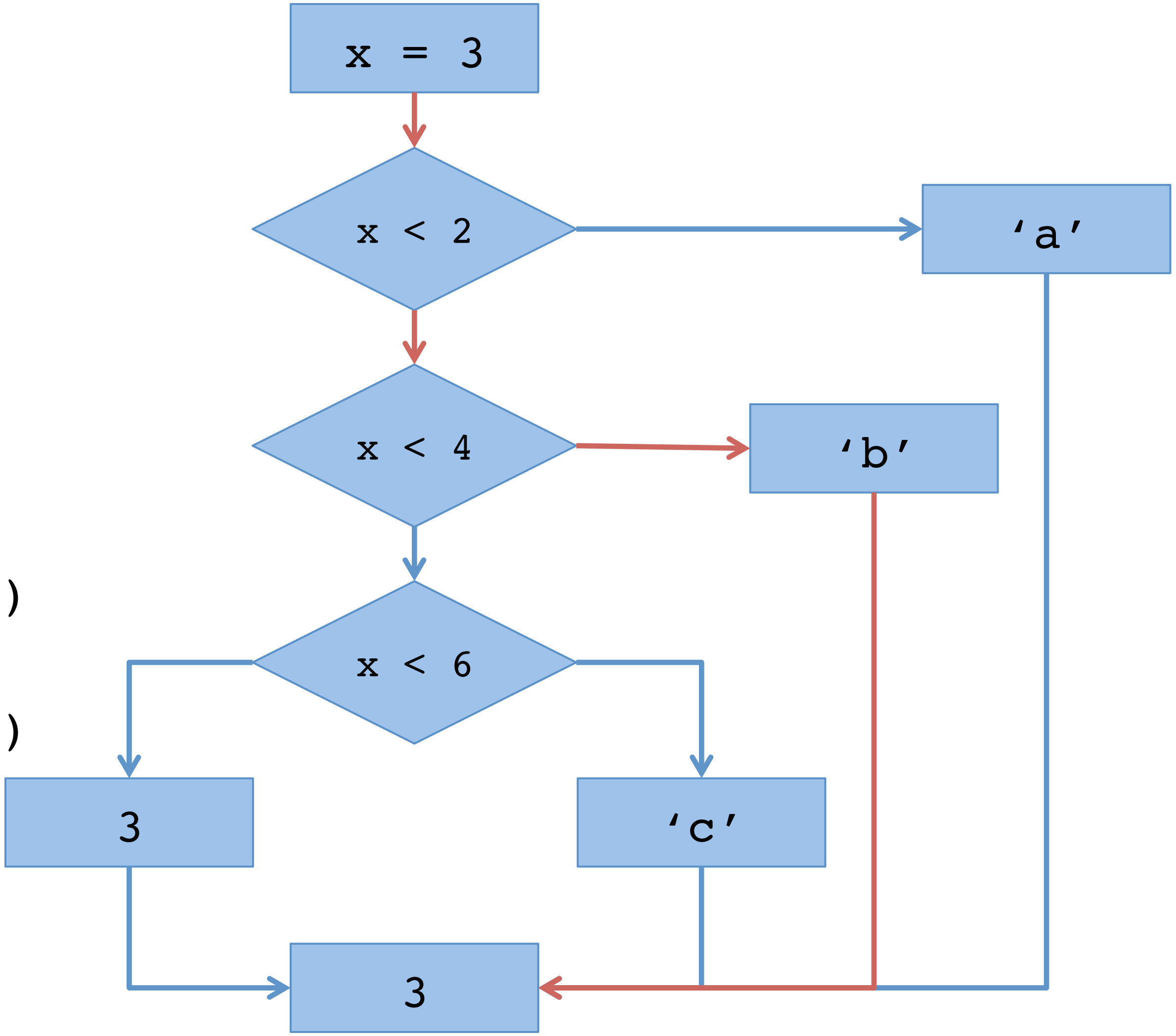
```
for i in range(n):
    for j in range(m):
        print(i, j)
print(‘Always run’)
```

```
if x < 2:
    if x < 0:
        print(x)
    else:
        print(‘inner else’)
else:
    print(‘outer else’)
print(‘Always run’)
```

Once inside the block, control skips the next block, *if it’s an else statements*

An extreme example

```
x = 3
if x < 2:
    print('a')
else:
    if x < 4:
        print('b')
    else:
        if x < 6:
            print('c')
        else:
            print('d')
print(x)
```



Wait, this is hairy

```
x = 3
if x < 2:
    print(‘a’)
else:
    if x < 4:
        print(‘b’)
    else:
        if x < 6:
            print(‘c’)
        else:
            print(‘d’)
print(x)
```

Maintaining block structure **and** keeping up with programming logic is error-prone (imagine if this continued for several more levels!)

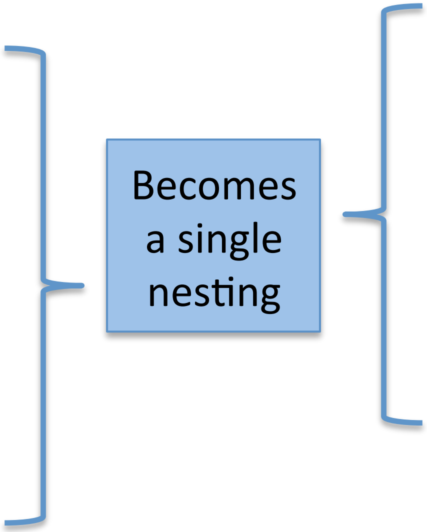
“else if”

- Sometimes decision making can get complicated
  - If you’re a freshman, do this
  - (But) if sophomore, do that
  - All others do this

```
if expression:
  block
elif expression:
  block
else:
  block
```

More clarity

```
x = 3
if x < 2:
    print(‘a’)
else:
    if x < 4:
        print(‘b’)
    else:
        if x < 6:
            print(‘c’)
        else:
            print(‘d’)
print(x)
```



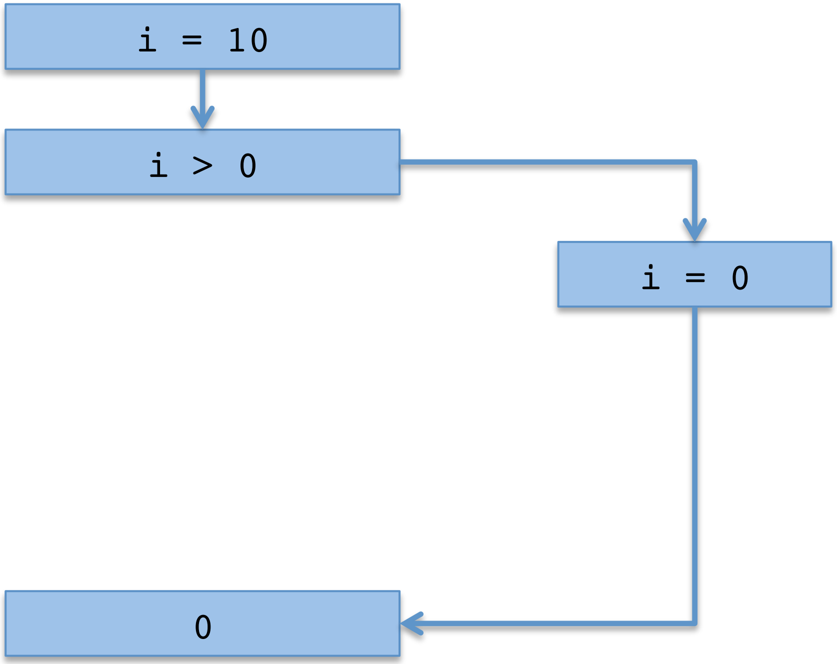
Becomes a single nesting

```
x = 3
if x < 2:
    print(‘a’)
elif x < 4:
    print(‘b’)
elif x < 6:
    print(‘c’)
else:
    print(‘d’)
print(x)
```

Recall if-statements

- Conditional code execution based on Boolean expressions

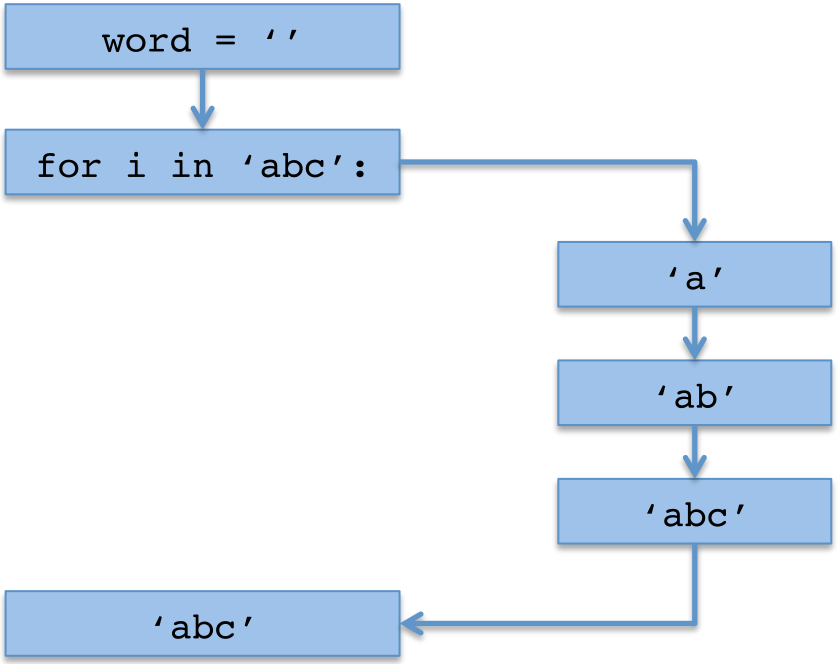
```
i = 10
if i > 0:
    i = 0
print(i)
```



Recall for-loops

- Repetitive code execution based on sequence data

```
word = ‘’
for i in ‘abc’:
    word = word + i
print(word)
```



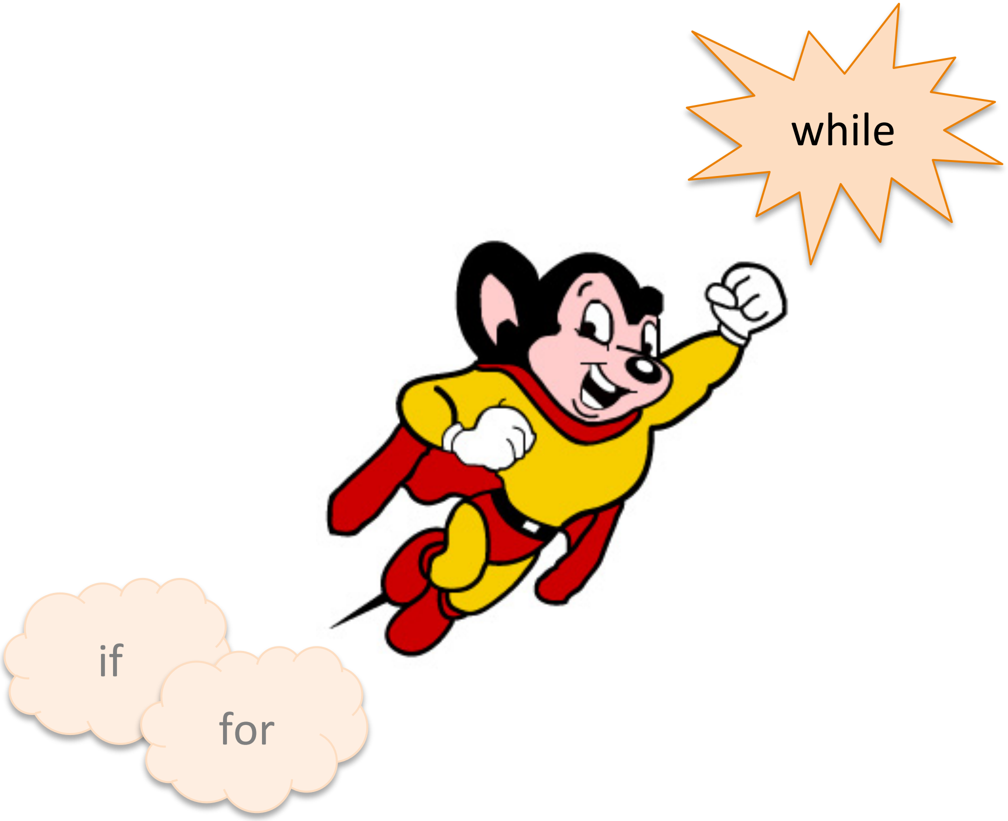
Combining forces

if-statements

- Boolean code execution

for-loops

- Repetitive structure



while

if

for

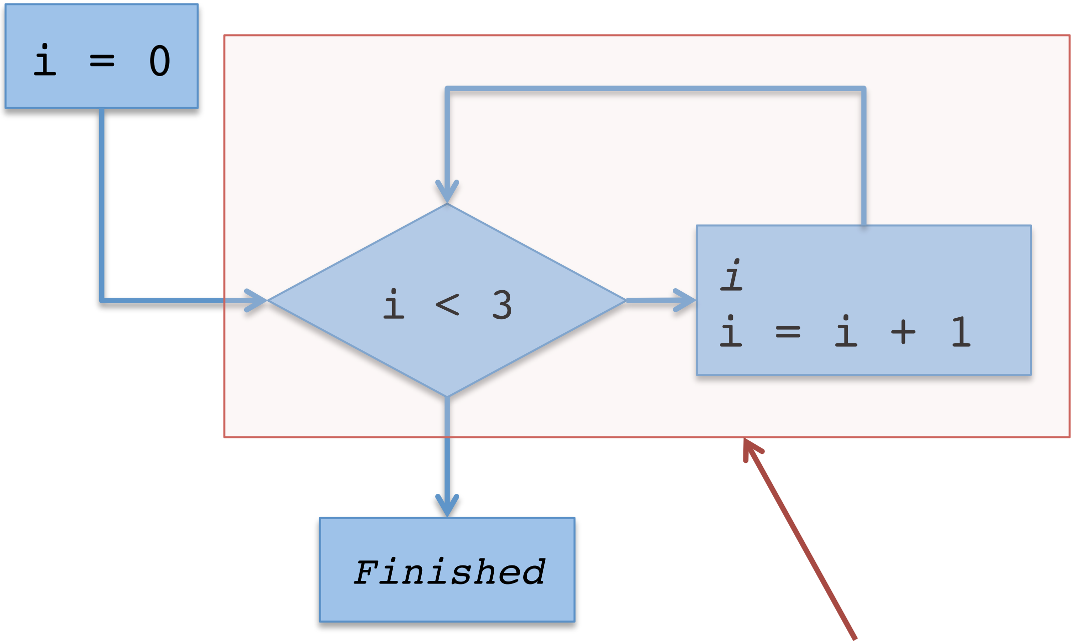
while loops

- A structure for repetition
- Controlled by a Boolean expression
- Closer to looping constructs of other languages

```
while expression:
  block
```

while loop in action

```
i = 0
while i < 3:
    print(i)
    i = i + 1
print('Finished')
```



Segment is run *while*
i is less-than three

Head to head showdown (part 1)

The while version

```
i = 0
while i < 3:
    print(i)
    i = i + 1
print(‘Finished’)
```

← Initialization →

← Update →

The for version

```
for i in range(3):
    print(i)

print(‘Finished’)
```

- Over static linear ranges, Python while- and for-loops can be used interchangeably
- for-loops have the advantage that iteration management is taken care of by the language

Not so fast!

- Loops are controlled by an iterator
  - When the iterator is finished, the loop is finished
- *Loops are also controlled by the programmer*
  - Programming constructs exist to alter these semantics

Taking control of the iteration

break

- Tell the interpreter to leave the loop immediately
- Move to the block that follows
  - Not necessarily the outer-most block!

continue

- Tell the interpreter to move on with the iteration
- Move to the top of the statement and start again
  - **for-loops**: moves the iterator to the next element in the sequence
  - **while-loops**: the iterator remains unaltered!

Example: continue

- Add 10 students to my directory
  - But don’t add Bob

```
i = 0
directory = []

while i < 10:
    stu = input(‘Name: ‘)
    if stu == ‘Bob’:
        continue
    directory.append(stu)
    i = i + 1

print(‘Finished’)
```

If the students name is Bob, control will move back to the beginning of the loop, *without incrementing i!*

Example: break

- Add 10 students to my directory
- If the user enters a particular keyword, stop adding students

```
i = 0
directory = []

while i < 10:
    stu = input(‘Name: ‘)
    if stu == ‘Stop’:
        break
    directory.append(stu)
    i = i + 1

print(‘Finished’)
```

If the user enters ‘Stop’, stop adding students immediately, *even if 10 students haven’t been entered yet*

Iteration control in for-loops

- Iteration constructs work in for-loops as well

```
for i in range(10):
    if (i == 5):
        break
    print(i)
```

Prints the first five numbers in the range

```
for i in range(10):
    if (i == 5):
        continue
    print(i)
```

Prints all numbers in the range, except 5

Head to head showdown (part 2)

The while version

```
while True:
    i = input(‘Name: ‘)
    if i == ‘’:
        break
print(‘Finished’)
```

The for version

```
???
```

- While-loops are advantageous when the number of iterations is not known before hand
  - The range isn’t static
- This is how most servers are implemented!

Expressions: a double edge sword

- The programmer controls the number of iterations
  - Unlike looping over sequences
- Must ensure the iterator is updated properly each time through the loop!

The good	• You don't have to iterate if you don't want to • You can control the speed, and direction, of the iteration
The bad	• You are forced to manage the iteration within the loop • Could forget to update all together (less common) • Update is based on complex decision structure (more common)
The ugly	• Getting iteration wrong can lead to unexpected results • Best case: the loop doesn't run as many times as desired • Worse case: the loop never ends!