

Conditional and Iterative Statements

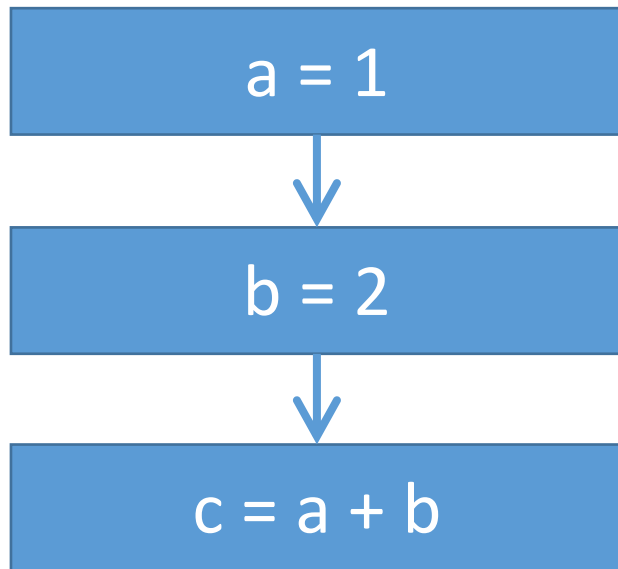
Dr. Sebastian Proksch

University of Zurich, Department of Informatics

Outline

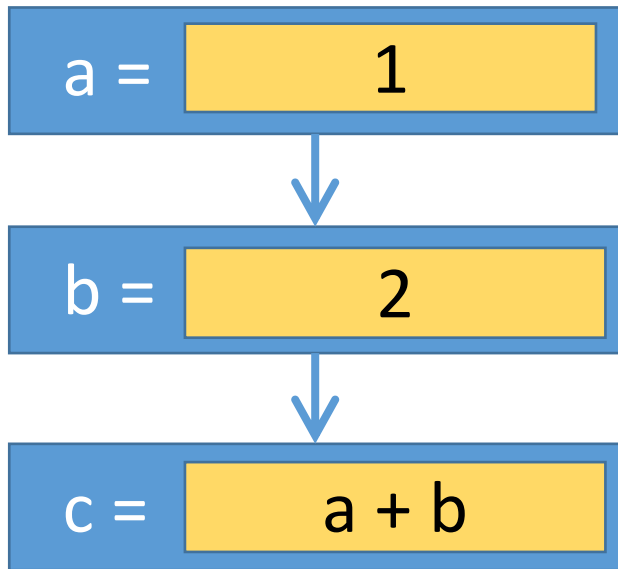
- Statements/Expressions
- Comparison Operators
- Conditional Control Flow
- Iterative Statements (“Loops”)
- Example Exercises

Statements



A Statement is the smallest element of a programming language that can stand alone.

Expressions



Expressions are typically part of statements. In Python, the boundary between both is fluent, because also expressions can stand alone.

An expression has a type and a value. This value might be None though.

Expression Types

literals

1, "a", True, ...

references

a, myVar, print, ...

boolean expressions ("bool")

a and b, a or b, not c, ...

arithmetic Expressions ("number")

1 + 2, 2 * a, c / 5, ...

... (many other)

Expressions Are Composable

Binary Expressions

```
«bool» or «bool» # -> «bool»  
«number» + «number» # -> «number»  
«number» > «number» # -> «bool»  
...
```

Unary Expressions

```
not «bool» # -> «bool»  
-«number» # -> «number»  
...
```

Ternary Expressions

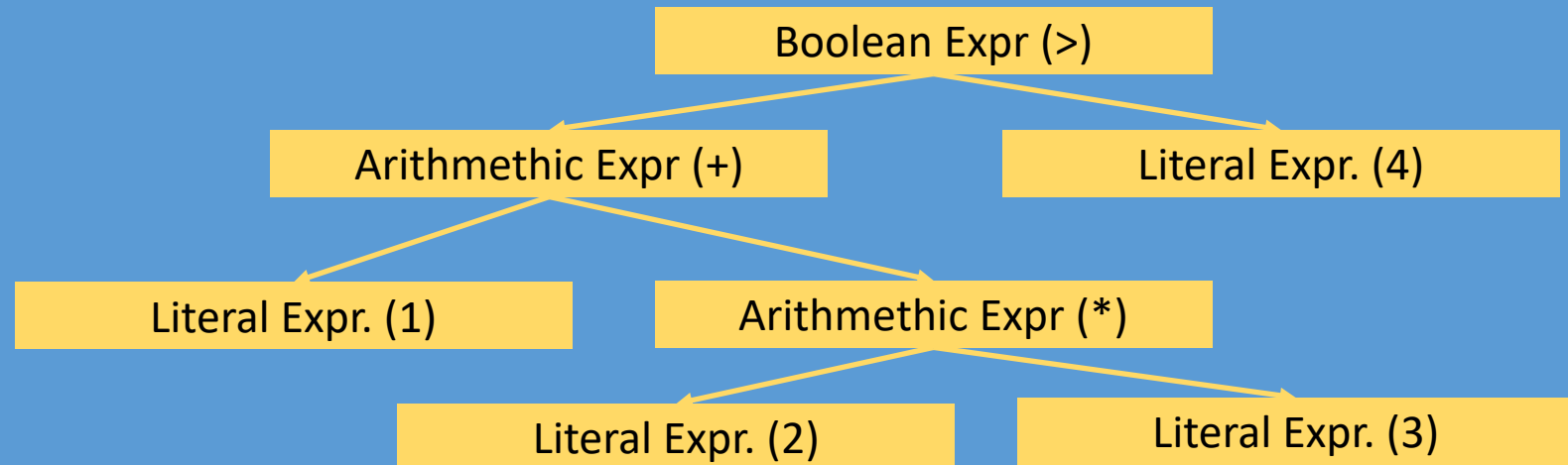
```
«expr» if «bool expr» else «expr» # -> «expr»
```

Every concrete expression has well-defined “holes”, a well-defined type, and a well-defined value

Expression Tree

```
print (1 + 2*3 > 4)
```

print



The operator precedence is always reflected in this expression tree.

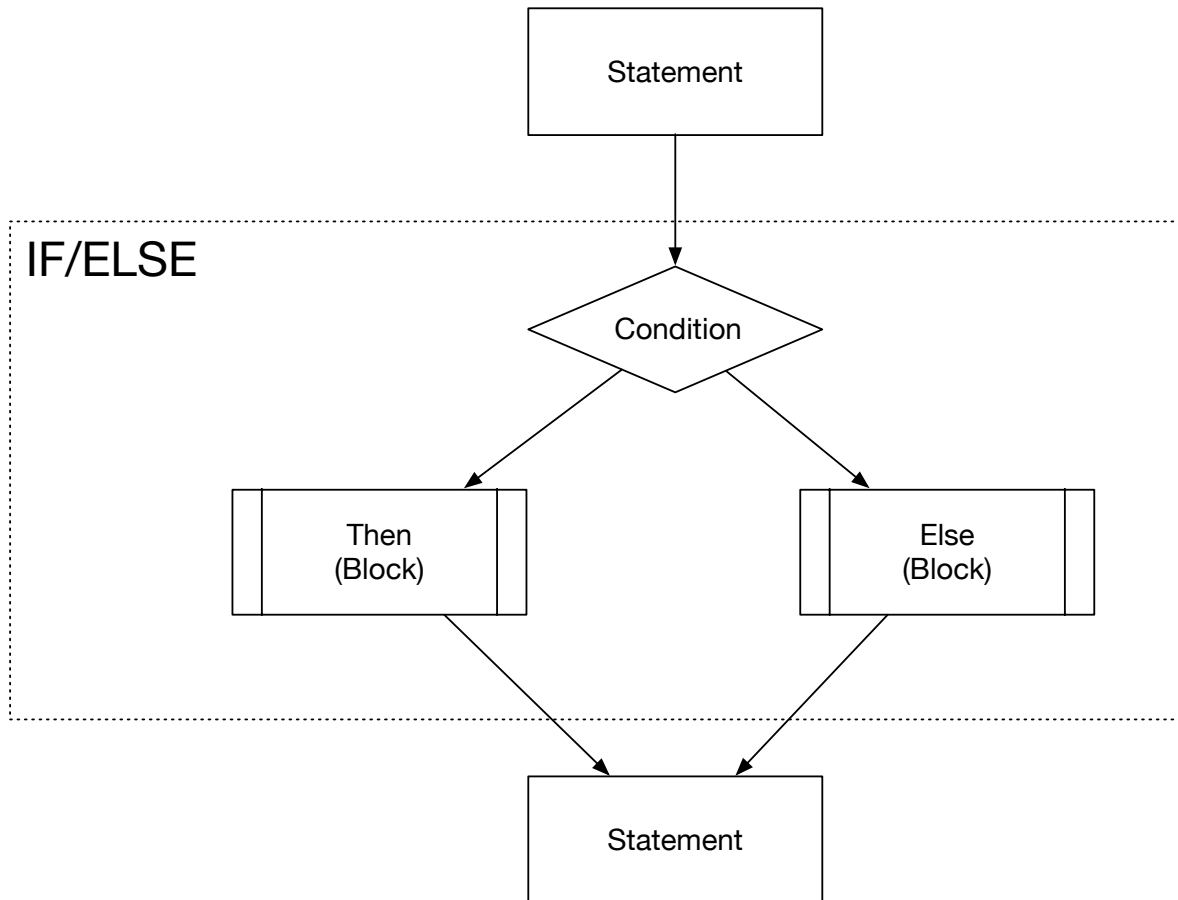
Conditional Statements

Overview of Comparison Operators

Operation	Result
<code>x == y</code>	returns True if x and y are equal, False otherwise
<code>x != y</code>	returns True if x and y are not equal, False otherwise
<code>x > y</code>	returns True if x is greater than y, False otherwise
<code>x < y</code>	returns True if x is less than y, False otherwise
<code>x >= y</code>	returns True if x is greater or equal than y, False otherwise
<code>x <= y</code>	returns True if x is less or equal than y, False otherwise

These operators are examples for conditions that might affect a program execution.

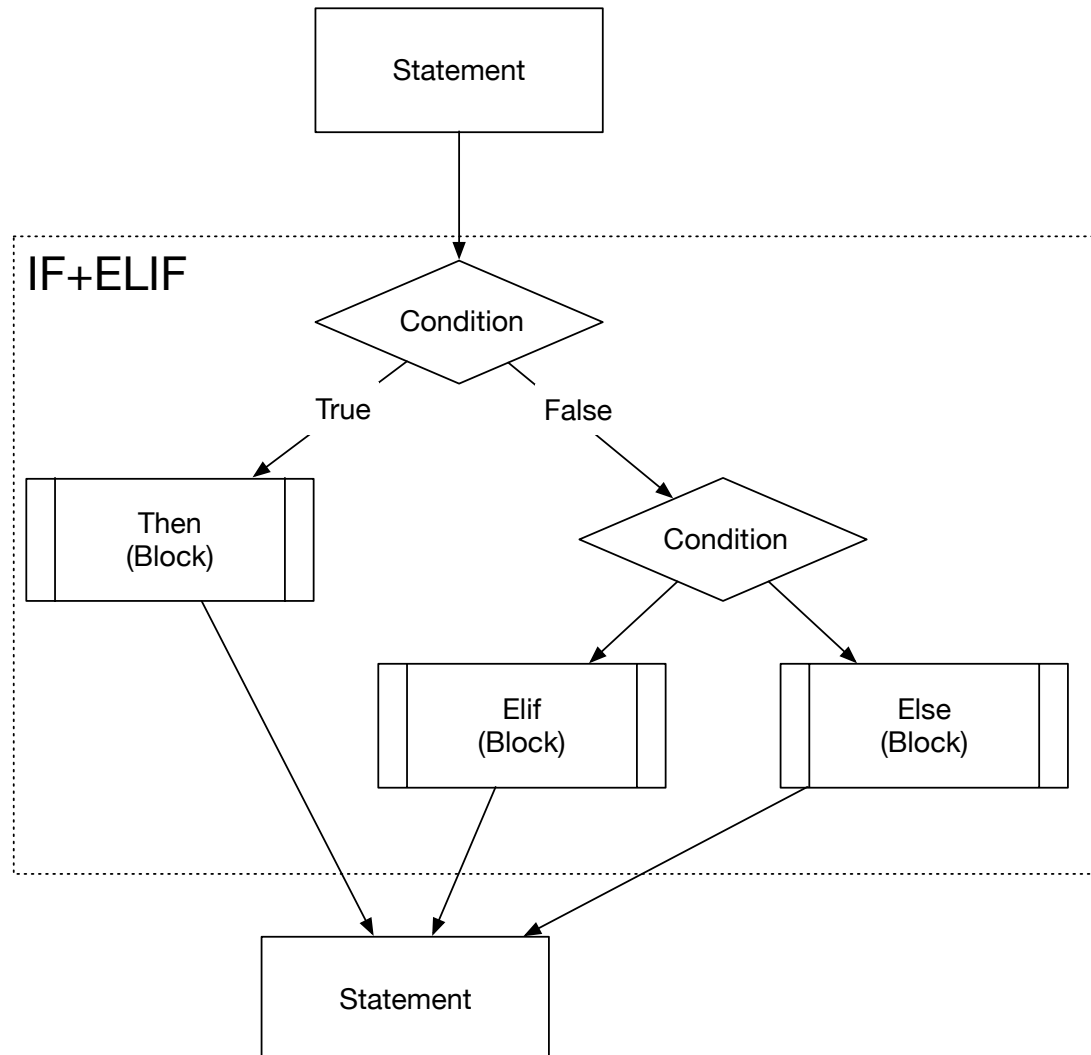
Conditional Statement



Depending on the condition, either the then block is being executed (True) or the else block (False).

```
if «bool»:  
    «statements»  
else:  
    «statements»
```

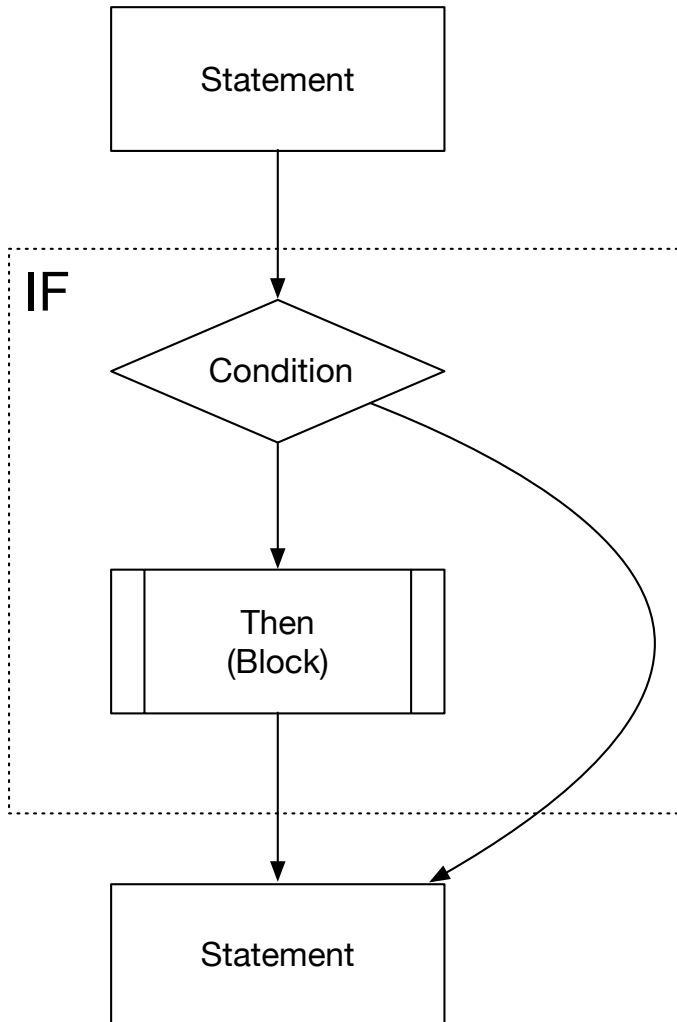
Conditional Statement With Multiple Conditions



An if can have multiple conditions. Only one is selected. If no condition applies, else is used as the fallback.

```
if «bool»:  
    «statements»  
elif «bool»:  
    «statements»  
else:  
    «statements»
```

Conditional Statement Without Else Case



The else block is optional and can be omitted. In this case, the then block is being executed when the condition holds (True).

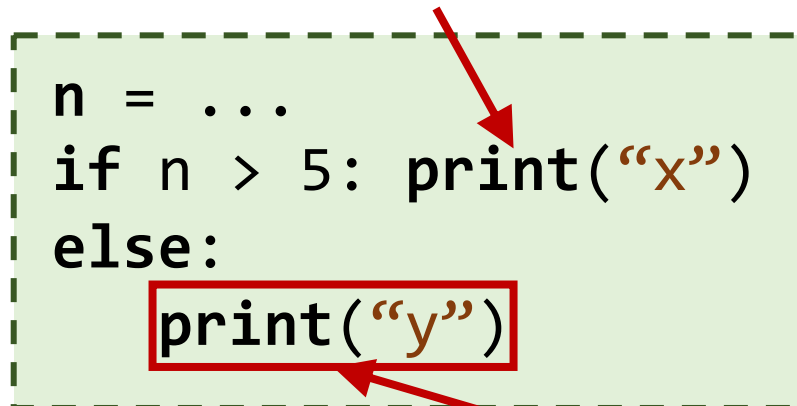
```
if «bool»:  
    «statements»
```

Blocks and Indentation

- Blocks can contain one or multiple statements.
- In Python, blocks are defined through indentation. Incorrect indentation will result in syntax errors. Other languages often use { . . . } to mark blocks.

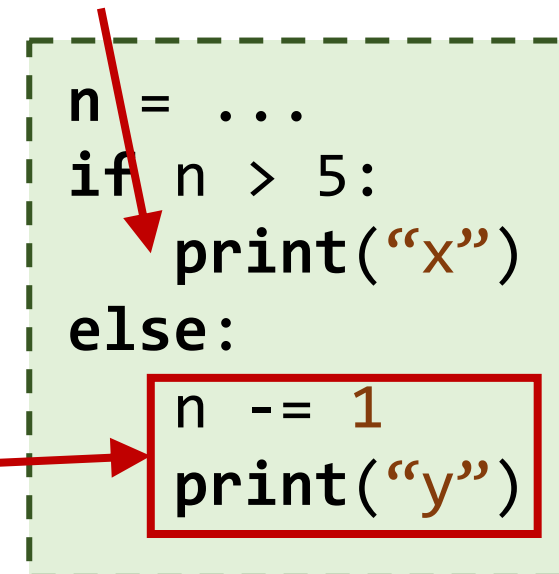
Single statements can be used inline...

```
n = ...  
if n > 5: print("x")  
else:  
    print("y")
```



or in a block.

```
n = ...  
if n > 5:  
    print("x")  
else:  
    n -= 1  
    print("y")
```



All subsequent statements with the same indentation form a block

Nested Blocks

```
n = ...  
if n > 5:  
    print("x")  
else:  
    if n < 0:  
        print("a")  
    else:  
        print("b")
```

Every time a statement can be inserted, it is also possible to start a block. In this example, multiple `if/else` blocks are nested in each other.

Conditional Expressions

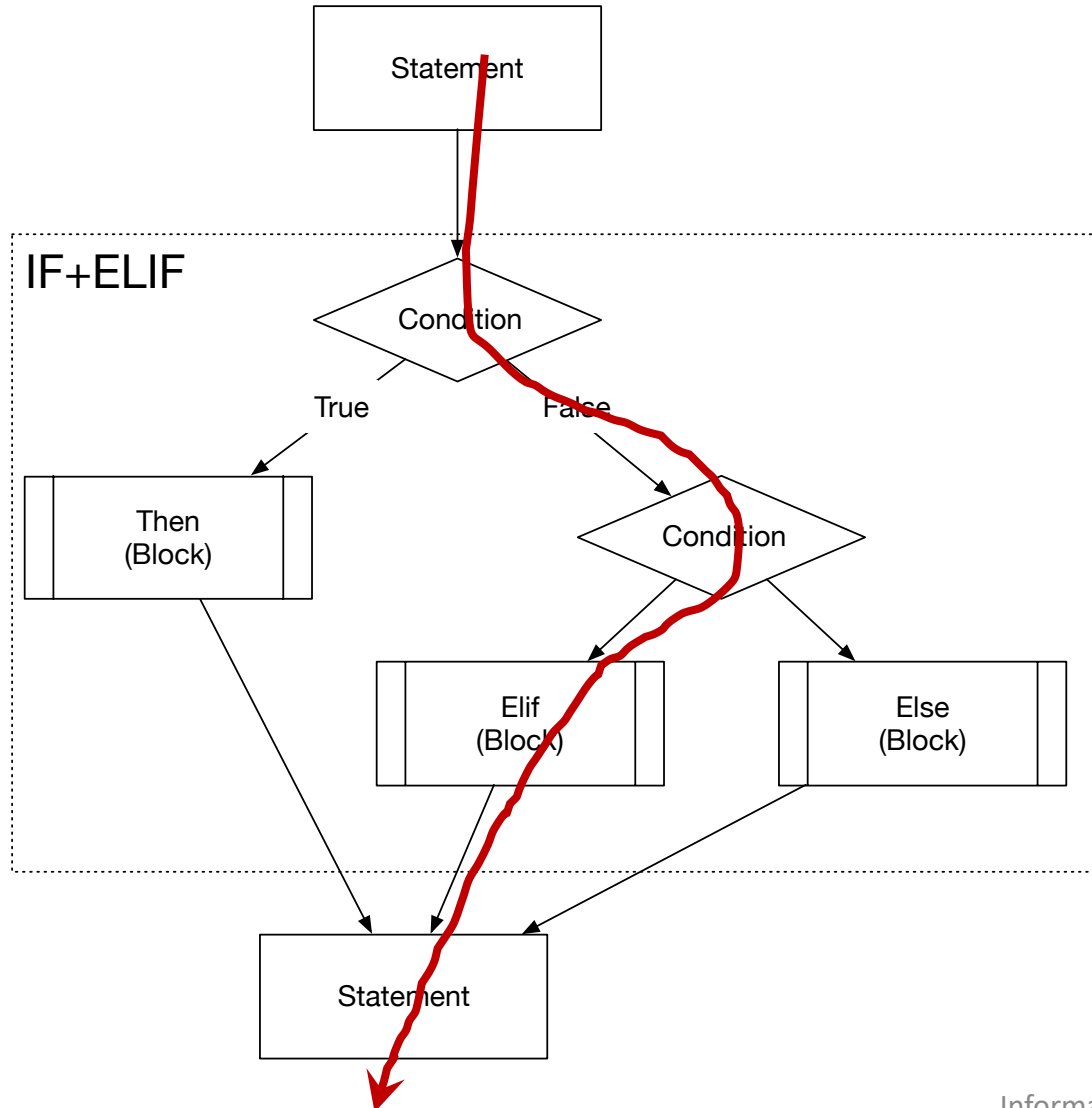
```
a, b = ...  
if a <= b:  
    minvalue = a  
else:  
    minvalue = b  
  
# Alternative  
minvalue = a if a <= b else b
```

The condition of the expression is evaluated first, then the expression to the left is evaluated if the expression is True otherwise the expression after else.

Control Flow

Control flow is the way the processor walks through the programming logic.

Control flow is deterministic, given the same values, another execution will follow the same path.



“False”

The following values are considered `False` by Python, everything else is considered `True`:

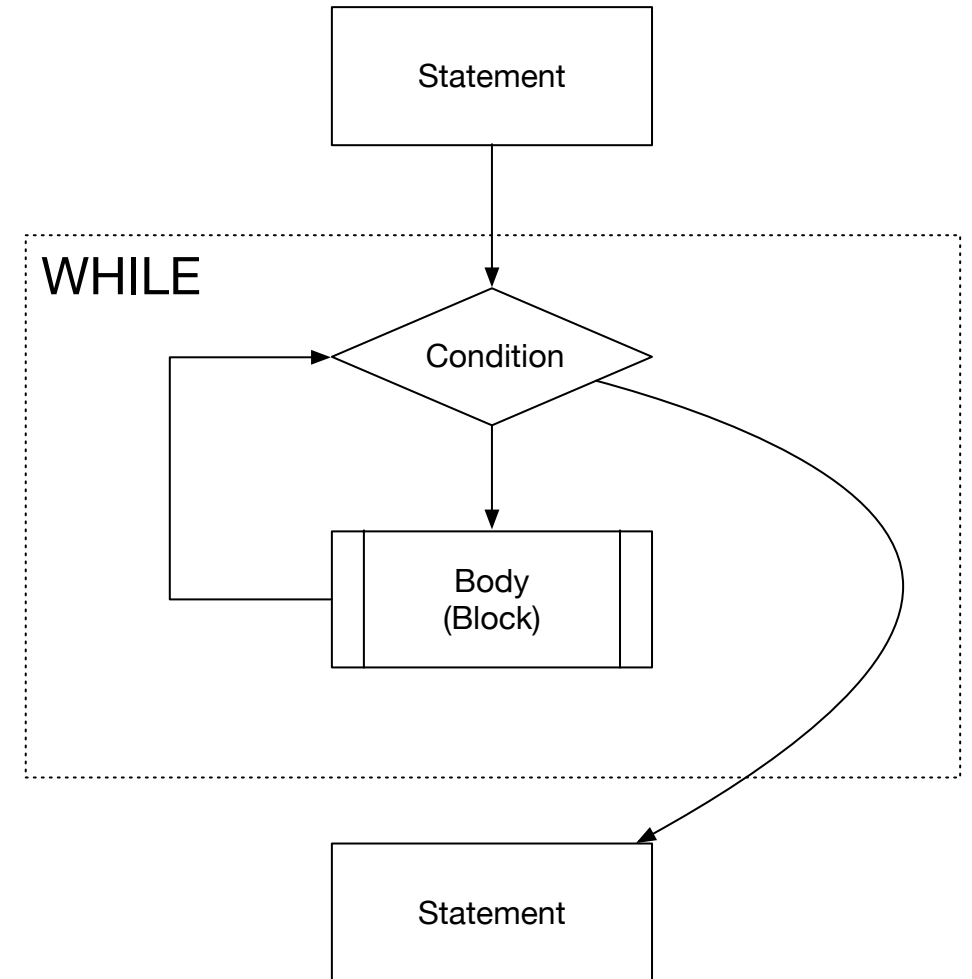
- `False` (a boolean expression)
- `None`
- `0`, `0L`, `0.0`, `0j`
- empty sequences: `""`, `()`, `[]` (will be covered later)
- empty mapping: `{}` (later...)
- Out of Scope: instances of user-defined classes with methods `__nonzero__()` and `__len__()` that return `0` or value `False`

Iterative Statements

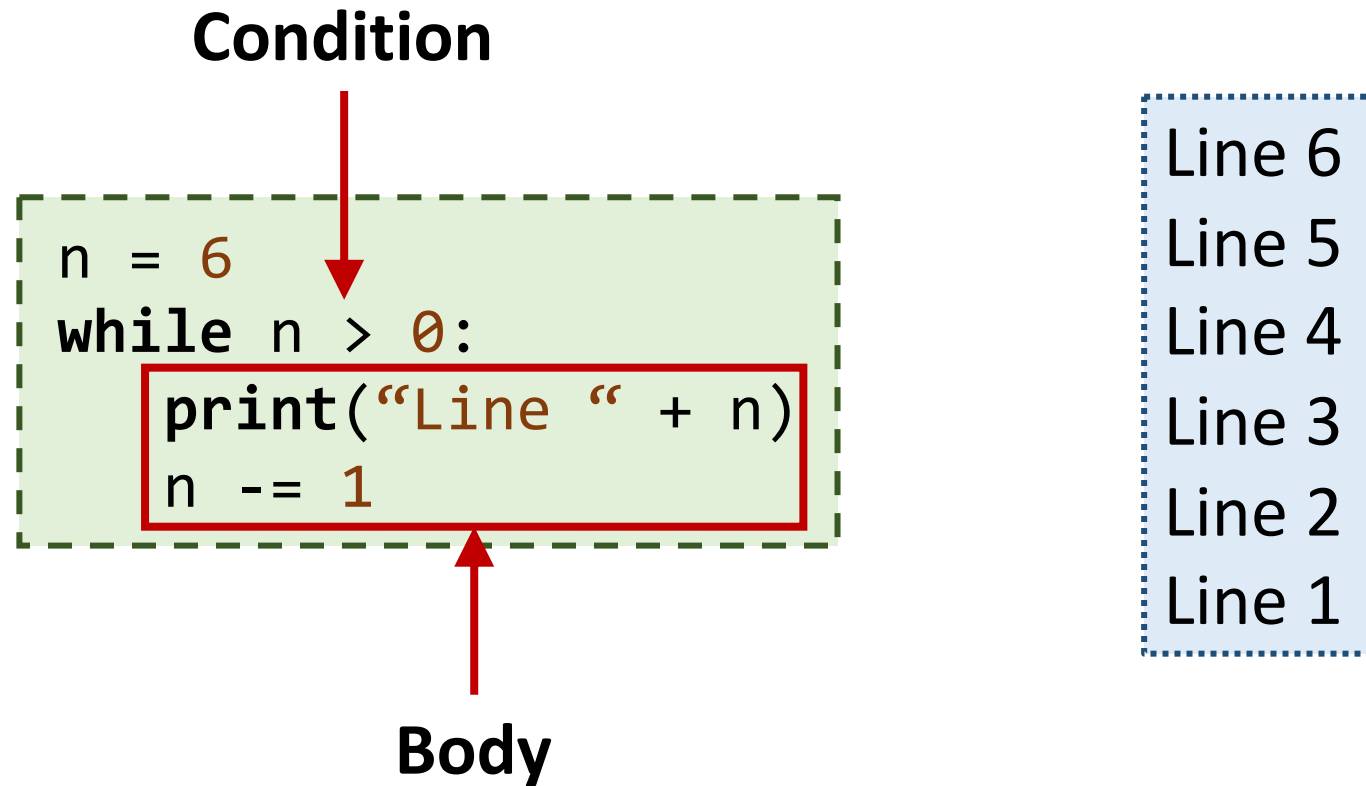
WHILE Loop

Control Flow Diagram

- the loop body is executed as long as the condition evaluates to `True`
- once it evaluates to `False` the rest of the code is executed
- if the condition is never `True`, the loop body will never be executed
- if the condition is always `True`, the loop will not terminate (“infinite loop”)



Python Example



Iterative Statements

FOR Loop

Excuse: range

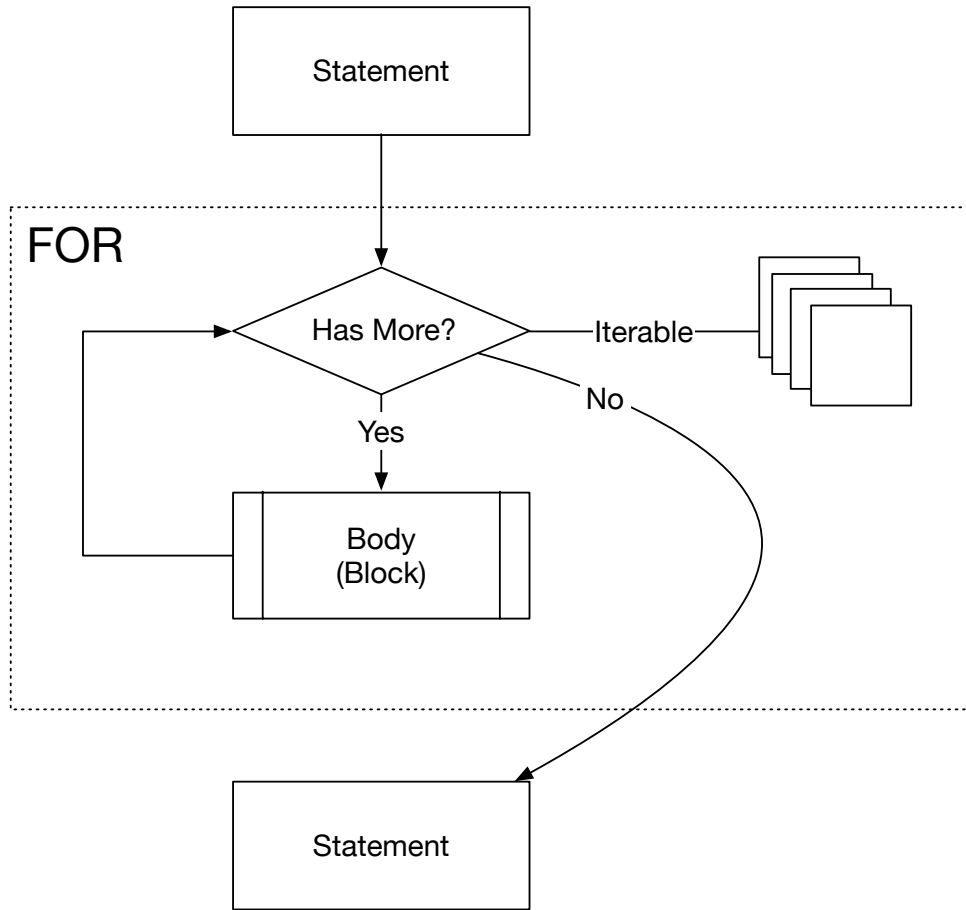
The function `range` can be used to generate an iterable sequence of numbers. It is often used in loops.

```
range([«start»,] «stop» [, «step»])
```

«stop» is mandatory, both «start» and «step» are optional.

```
range(2)           # 0, 1
range(0, 2, 1)      # 0, 1
range(2, 6)         # 2, 3, 4, 5
range(1, 8, 3)       # 1, 4, 7
range(3, 1, -1)     # 3, 2
```

Control Flow Diagram



The for loop will iterate over all available elements and bind them to a variable that is available in the loop body.

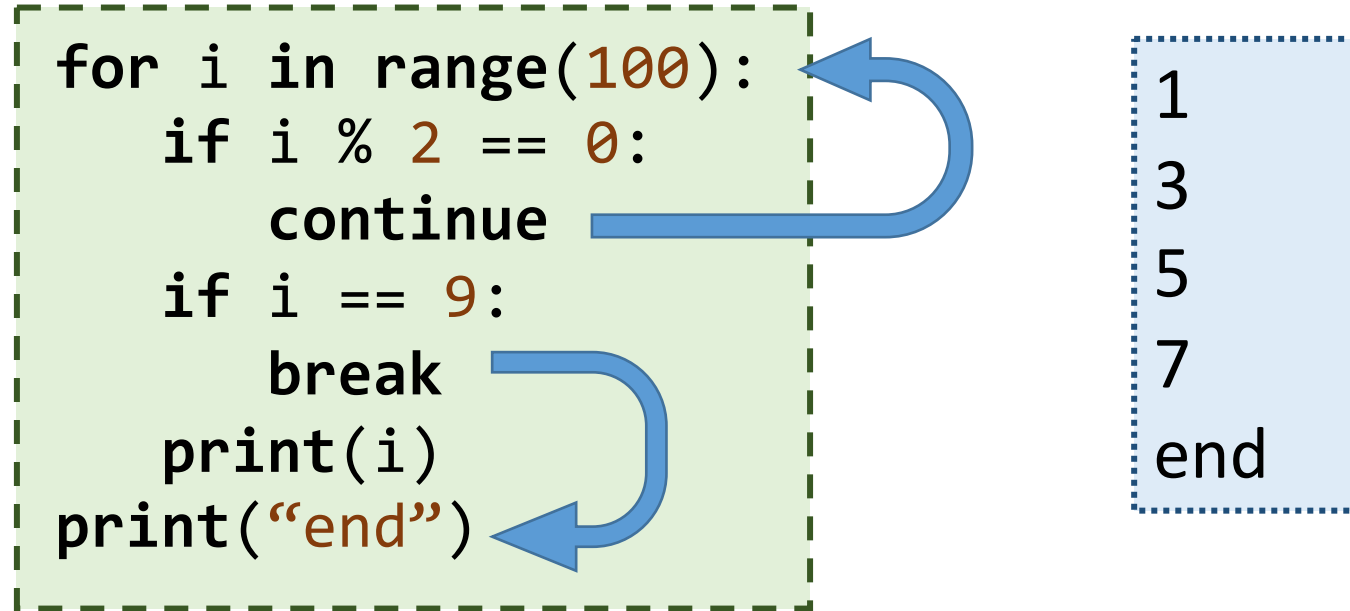
```
it = range(6)
for n in it:
    print("Line " + n)
```

Line 0
Line 1
Line 2
Line 3
Line 4
Line 5

Iterative Statements

Advanced usage of loops...

Break & Continue



The two statements `continue` and `break` can be used to control the control flow inside a loop. Use `continue` to skip to the next iteration, use `break` to cancel the whole loop.

Nested Loops

Loops are regular statements,
so -of course- they can be used
in nested blocks.

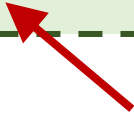
```
for i in range(4):  
    print("={}= ".format(i))  
    for j in range(i):  
        print(j)
```

```
=0=  
=1=  
0  
=2=  
0  
1  
=3=  
0  
1  
2
```

Beware of Infinite Loops

Make sure that a loop terminates!
It is very easy to accidentally write
code that runs forever.

```
n = 6  
while n:  
    print(n)  
    n-=1
```



What happens if you forget the decrement?

More About Strings

Strings Offer Many Convenient Operations

```
s = "some string..."
s.startswith("some") # Bool: True
s.endswith("!") # Bool: False
"xxx" in s # Bool: False
"1".isdigit() # Bool: True
"e".isupper() # Bool: False
"r".islower() # Bool: True
```

All of these boolean operators can be used as conditions in if statements or while loops.

String are Sequences and Can Be Iterated

```
idx = 1
for c in "lalelu":
    print("Char {} is {}".format(idx, c))
    idx += 1

print("The string has length {}".format(len(s)))
```

Exercises

Exercise 0: if __name__ ?

- Why do all exercise scripts in ACCESS start with this weird construct?

Exercise 1

- Compute the sum of the first n numbers, where n is a value provided by the user.
- Try to come up with two solutions: one should be using while and the other one should be based on a for loop.

Exercise 2

- Compute the sum of the first n even numbers.

Exercise 3

Assign a random value to a variable and ask the user to guess the value. If the guess is lower than the chosen value, print 'too low', if it is greater, print 'too high'. In case the number is correct, 'correct' and terminate the program.

You can use the following snippet to generate a random number:

```
import random
rnd = random.randint(«lower bound», «upper bound»)
# «lower bound» <= rnd <= «upper bound»
```

Exercise 4

- Write a program that can calculate the average from a list of grades. The program should ask the user to provide the grades one by one. A grade of -1 should end the program and print out the average of the grades.