



# 15-110 PRINCIPLES OF COMPUTING – F18

## LECTURE 5:

## STRING OPERATIONS, BOOLEAN TYPES, CONDITIONALS

TEACHER:

GIANNI A. DI CARO

# Strings so far

- **String instantiation**

```
greet_joe = "Hello Joe"
greet_mary = 'Hello Mary'
greet_bob = '''Hello Bob'''
greet_eve = """Hello Eve"""
greet_all = '''Hello Joe and Mary and
              Bob and Eve'''
```

- **String concatenation, + overloaded operator**

```
greet_joe = "Hello Joe"
greet_mary = "hello Mary"
print(greet_joe + ", " + greet_mary)
print (greet_joe + 2)
```

- **String duplication, \* overloaded operator**

```
zero_bit = "0"
one_bit = '1'
binary_256 = one_bit + (8 * zero_bit)
print("256 in binary format is", binary_256, binary_256 * (-4))
```

- **Indexing:**

```
greet="Hello Joe"
print(greet[0], greet[4])
print(greet[-4])
```

H	e	l	l	o		J	o	e
0	1	2	3	4	5	6	7	8

H	e	l	l	o		J	o	e
-9	-8	-7	-6	-5	-4	-3	-2	-1

# String operators: Membership

- **Part of, `in` operator, overloaded:** Membership operator that returns `True` if the first operand is contained within the second, and `False` otherwise

```
s = "Joe"  
in_hello = s in "Hello Joe"  
in_food = s in "Yummy meal"  
print(in_hello, in_food, type(in_hello))
```

```
True False <class 'bool'>
```

- **Not Part of, `not in` operator, overloaded:** Membership operator that returns `True` if the first operand is not contained within the second, and `False` otherwise

```
s = "Joe"  
in_hello = s not in "Hello Joe"  
in_food = s not in "Yummy meal"  
print(in_hello, in_food, type(in_hello))
```

```
False True <class 'bool'>
```

# Built-in String functions: Length and Casting

---

- **len()** function: Returns the length of a string

```
s = "Joe and Mary"  
length = len(s)  
print(s, length)
```

```
Joe and Mary 12
```

- **str(o)** function: *Virtually any object obj in Python can be rendered as a string.* Therefore, `str(obj)` returns the string representation of a python object `obj`:

```
n = 150.5  
s = str(n)  
print(s)
```

```
150.5
```

# Built-in String functions: Integer code of a character

---

- **ord(c) function:** Takes a character `c` and returns the *Unicode* UTF-8 code

```
n1 = ord('a')  
n2 = ord('€')  
print(n1, n2)
```

```
97 8364
```

- **chr(n) function:** It does the opposite, it returns a character value for the given integer `n` representing a UTF-8 code (1,114,112 is the max number of code points available in UTF-8)

```
s1 = chr(97)  
s2 = chr(8364)  
print(s1, s2)
```

```
a €
```

# Immutability/Mutability and Identity of an object

- Strings are **immutable** object types: we can't change the value of a string literal / variable!
  - Once created, the string keeps its value for its entire lifetime

```
s = "abcd"  
print( s[0]+s[3])  
s[3] = 'z'
```

```
ad  
TypeError: 'str' object does not support item assignment
```

- We can “change” the value of a string variable by *reassigning* its value, which amounts to create a new string variable

```
s = "abcd"  
print( s[0]+s[3])  
s = "abcz"  
print(s)
```

```
ad  
abcz
```

- This clearly holds for all scalar object types, which all are *immutable* (a change in the value corresponds into a new variable, possibly a new memory location)

# Immutability/Mutability and Identity of an object

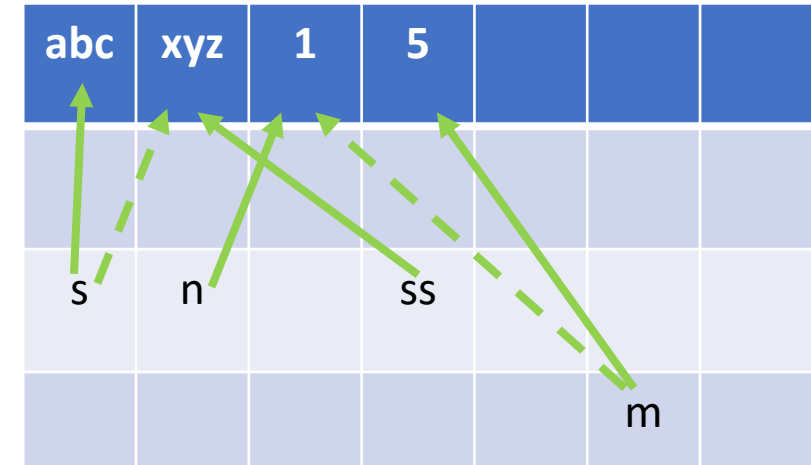
- **`id(obj)` function:** Takes an object `obj` as an input and returns its **identity**, an integer number that is unique for the object during its lifetime in the program
  - For an *immutable* object type, you can think of the identity as the address in memory of the literal(s) that give the *value of the object*
  - For a *mutable* object type, you can think of the identity as the address in memory of the variable

```
s = "abc"
ss = "xyz"
n = 1
m = 5
print( id(s), id(ss), id(n), id(m))
```

```
4652072720 4327641640 4307354688 4307354816
```

```
s = "xyz"
ss = "xyz"
n = 1
m = n
print( id(s), id(ss), id(n), id(m), id(1), id("abcd"))
```

```
4652072720 4652072720 4307354688 4307354688
4307354688 4652072720
```



# String slicing

- **Extracting substrings from a string**, known as string slicing. If `s` is a string, an expression of the form `s[m:n]` returns the portion of `s` starting with position `m`, and up to but not including position `n`

```
s = "Hello Joe"  
ss = s[0:4]  
print(ss)
```

Hell

- Why isn't the character at position `n` not included?
  - The result is a string of `m-n` characters
- Shortcuts: omitting indices produces some default behavior
  - `s[:4]` is equivalent to `s[0:4]`
  - `s[2:9]` is equivalent to `s[2:]` and to `s[2:len(s)]`
  - `s[:]` is equivalent to `s` (it's precisely the same object in memory, same id)
- Slicing can be used to modify a string, by **creating a new variable** including the desired changes:
  - `s_new = s[0:5] + '! ' + s[6:] + ' and Mary'`
- Slicing into empty strings
  - `s[4:2]` returns the empty string
  - `s[2:2]` returns the empty string
- Slicing with negative indices
  - `s[0:4]` and `s[-9:-5]` return the same substring



# String slicing with a stride

- **Extracting substrings of non (necessarily) adjacent characters from a string**, known as slicing with a stride: If `s` is a string, an expression of the form `s[m:n:s]` returns the portion of `s` starting with position `m`, and up to but not including position `n`, with the third index `s` designating a stride (a step), which indicates how many characters to jump after retrieving each next character in the slice

```
s = "Hello Joe"  
ss = s[0:9:2]  
print(ss)
```

HloJe

Slice 0:9:2 starts with the first character and ends with the last character (the whole string), and every second character is skipped

- Indices can be omitted: first and second indices default to the first and last characters respectively, while the third defaults to 1
  - `s[::4]` is equivalent to `s[0:9:4]`
  - `s[:6:2]` is equivalent to `s[0:6:2]`
  - `s[1:6:]` is equivalent to `s[1:6:1]`
  - `s[::]` is equivalent to `s` (it's the same object)
- Slicing with negative indices: steps are *backward*, with the first index that must be greater than the second index
  - `s[4:0:-1]` gives 'olle'
  - `s[::-1]` gives 'eoJ olleH'

# Built-in String *Methods*

---

- **Functions:** callable procedures that can be invoked to perform specific tasks (can take generic arguments)
- **Method:** a specialized type of callable procedure that is tightly associated with an object. Like a function, a method is called to perform a precise task, but it is invoked on a specific object and has knowledge of its target object during execution → Will understand this more when study class objects
- **Dot notation** for invoking a method on an object:

`obj.method_name(parameters)`

```
s = "Hello Joe"  
ss = s.lower()  
print(ss)
```

hello joe

```
s = "Hello Joe"  
ss = s.upper()  
print(ss)
```

HELLO JOE

Let's look at the methods built-in with the string (class) type, and let `s` be a string variable

# Built-in String Methods: Case Conversion

---

- `s.capitalize()` : returns a copy of `s` with the first character converted to uppercase and all other characters converted to lowercase:
- `s.swapcase()` : returns a copy of `s` with uppercase alphabetic characters converted to lowercase and vice versa. Non-alphabetic characters are unchanged.
- `s.title()` : returns a copy of `s` in which the first letter of each word (separated by spaces) is converted to uppercase and remaining letters are lowercase
- `s.lower()`
- `s.upper()`

# Built-in String Methods: Count, Find, and Replace

---

- `s.count(<sub>)` : returns the number of non-overlapping occurrences of substring `<sub>` in `s`
  - `s = "moo ooh pooh"`
  - `s.count("oo") → 3`
  - `"moo ooh pooh".count("oo")`
- `s.count(<sub>, <start>, <end>)` : returns the number of non-overlapping occurrences of substring `<sub>` in the `s` slice defined by `<start>` and `<end>`
  - `s = "moo ooh pooh"`
  - `s.count("oo", 3, len(s)) → 2`

# Built-in String Methods: Count, Find, and Replace

---

- `s.endswith(<suffix>)` : returns True if s ends with the specified <suffix>, and False otherwise
  - `s = "crazy bar"`
  - `s.endswith("bar") → True`
- `s.endswith(<suffix>, <start>, <end>)` : as above, but now the comparison is restricted to the substring indicated by <start> and <end>
  - `s = "crazy bar"`
  - `s.endswith("bar", 0, 5) → False`
- `s.startswith(<suffix>)` : analogous to `endswith()`, but checking if the string begins with a given substring

# Built-in String Methods: Count, Find, and Replace

---

- `s.find(<sub>)`: returns the lowest index in `s` where the substring `<sub>` is found, -1 is returned if the substring is not found
  - `s = "crazy bar bar"`
  - `s.find("bar") → 6`
  - `s.find("star") → -1`
- `s.find(<suffix>, <start>, <end>)`: as above, but now the search is restricted to the substring indicated by `<start>` and `<end>`
  - `s = "crazy bar bar"`
  - `s.find("bar", 7, 13) → 10`

# Built-in String Methods: Count, Find, and Replace

---

- `s.rfind(<sub>)`: searches `s` starting from the end, such that it returns the highest index in `s` where the substring `<sub>` is found, -1 is returned if the substring is not found
  - `s = "crazy bar bar"`
  - `s.rfind("bar") → 10`
  - `s.find("bar") → 6`
- `s.rfind(<suffix>, <start>, <end>)`: as above, but now the search is restricted to the substring indicated by `<start>` and `<end>`
  - `s = "crazy bar bar"`
  - `s.rfind("bar", 0, 10) → 6`

# Built-in String Methods: Count, Find, and Replace

---

- `s.replace(<old>, <new>)`: returns a *copy* of `s` with all occurrences of substring `<old>` replaced by `new`. If there are no occurrence of `<old>`, the copy is identical to the original (but it's still a different object)
  - `s = "one step, two steps, three steps"`
  - `s.replace("step", "stop")` → "one stop, two stops, three stops"
- `s.replace(<old>, <new>, <max>)`: as above, but now the number of replacements is limited to the `<max>` value
  - `s = "one step, two steps, three steps"`
  - `s.replace("step", "stop", 2)` → "one stop, two stops, three steps"



# Built-in String Methods: String formatting

---

- `s.center(<width>[, <fill>])`
- `s.expandtabs(tabsize=8)`
- `s.ljust(<width>[, <fill>])`
- `s.lstrip([<chars>])`
- `s.rjust(<width>[, <fill>])`
- `s.rstrip([<chars>])`
- `s.strip([<chars>])`
- `s.zfill(<width>)`

# Built-in String Methods: Character classification

---

- `s.isalpha()`
- `s.isalnum()`
- `s.isdigit()`
- `s.isidentifier()`
- `s.islower()`
- `s.isupper()`
- `s.isprintable()`
- `s.isspace()`
- `s.istitle()`

# String formatting using escape sequences

---

- `print("He said, "What's there?")` → `SyntaxError: Invalid syntax`
- `print('He said, "What's there?")` → `SyntaxError: Invalid syntax`
- `print("""He said, "What's there?""")` → Ok!
- Alternative way: Use of **Escape sequences**
  - An escape sequence **starts with a backslash** `\` such that what follows is interpreted differently from usual
  - `print("He said, "What\'s there?")` → Ok
  - `print('He said, \'What\'s there?')` → Ok
- `\n` : new line feed is inserted `print("Hello!\nThis goes on a new line")`
- `\t` : tabular space is inserted `print("Hello!\t\tThis gets two tab spaces")`
- `\\` : this allows to write file/folder paths in windows `print("C:\\Python64\\Lib")`
- `\a` : this rings a bell! `print("This rings a bell\a")`

# String formatting using escape sequences

Escape Sequence	Description
<code>\newline</code>	Backslash and newline ignored
<code>\\</code>	Backslash
<code>\'</code>	Single quote
<code>\"</code>	Double quote
<code>\a</code>	ASCII Bell
<code>\b</code>	ASCII Backspace
<code>\f</code>	ASCII Formfeed
<code>\n</code>	ASCII Linefeed
<code>\r</code>	ASCII Carriage Return
<code>\t</code>	ASCII Horizontal Tab
<code>\v</code>	ASCII Vertical Tab
<code>\ooo</code>	Character with octal value ooo
<code>\xHH</code>	Character with hexadecimal value HH

# Boolean types and Comparison (Relational) operators

---

- **bool: Boolean (logical) values**
  - Instances of literals of type `bool` are: `True`, `False`
  - `x = True` defines a boolean variable with a true value
  - `print(x) → True`
- A **boolean expression** is an expression that evaluates to a boolean value, true or false
- A boolean expression results from the application of **comparison operators**
  - `x == y`    *x is equal to y*
  - `x != y`    *x is not equal to y*
  - `x > y`    *x is greater than y*
  - `x < y`    *x is less than y*
  - `x >= y`    *x is greater than or equal to y*
  - `x <= y`    *x is less than or equal to y*

x,y are expressions that can evaluate to numbers, strings, boolean types,...  
overloaded operators

# Boolean types and Comparison (Relational) operators

```
n1 = 5
n2 = 7
d = n2 == n1
print(d, type(d))

n1 = 5.5
n2 = 7.1
d = n2 == n1
print(d, type(d))
```

```
n1 = 5
n2 = 7
d = n2 != n1
print(d, type(d))

n1 = 5.5
n2 = 7.1
d = n2 != n1
print(d, type(d))
```

16 different examples  
using relational operators

```
s1 = 'b'
s2 = 'c'
d = s2 == s1
print(d, type(d))

s1 = True
s2 = False
d = s2 == s1
print(d, type(d))
```

```
s1 = 'b'
s2 = 'c'
d = s2 != s1
print(d, type(d))

s1 = True
s2 = False
d = s2 != s1
print(d, type(d))
```

```
n1 = 5
n2 = 7
d = n2 > n1
print(d, type(d))

n1 = 5.5
n2 = 7.1
d = n2 > n1
print(d, type(d))
```

```
n1 = 5
n2 = 7
d = n2 >= n1
print(d, type(d))

n1 = 5.5
n2 = 5.5
d = n2 >= n1
print(d, type(d))
```

```
s1 = 'b'
s2 = 'c'
d = s2 > s1
print(d, type(d))

s1 = True
s2 = False
d = s2 > s1
print(d, type(d))
```

```
s1 = 'bass'
s2 = 'class'
d = s2 >= s1
print(d, type(d))

s1 = True
s2 = False
d = s2 >= s1
print(d, type(d))
```

# Boolean types and Logical operators: and

---

- **and:** `(x and y)` evaluates to `True` if and only if both `x` and `y` are `True` expressions
  - `a = ( (2 != 3) and ('yes' == 'yes')) → True`
  - `a = ( (2 != 2) and ('yes' == 'yes')) → False`
  - `a = ( (2 != 3) and ('yes' == 'no')) → False`
  - `a = ( (2 != 2) and ('yes' == 'no')) → False`
- Example of typical application: check whether a value `x` belongs or not to a certain interval
  - is  $0 \leq x \leq 5$ ?  
`range = (x >= 0) and (x <= 5)`
- Example of typical application: guarantee that two conditions are both satisfied
  - is battery more than 95% and the phone is on?  
`conditions = (battery >= 0.95) and (phone == "on")`

# Boolean types and Logical operators: and

- **and**: `(x and y)` evaluates to `True` if and only if both `x` and `y` are `True` expressions

- `a = (2 != 3) and ('yes' == 'yes')) → True`
- `a = (2 != 2) and ('yes' == 'yes')) → False`
- `a = (2 != 3) and ('yes' == 'no')) → False`
- `a = (2 != 2) and ('yes' == 'no')) → False`

Four cases, for all possible combinations of truth values of two operands, `p` and `q`

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

- 0 for False
- 1 for True

A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

**AND**

`x = True`

`y = False`

`a = x * y`

`print(a, type(a)) → False <class 'int'>`

`print(x*x, type(x*x)) → True <class 'int'>`

`print(x*5, type(x*5)) → 5 <class 'int'>`

`print(x*5.6, type(x*5.6)) → 5.6 <class 'float'>`

Logical truth  
table for AND

- ✓ **`*`, overloaded operator (but result changes type)**
- ✓ Logical and is equivalent multiplication



# Boolean types and Logical operators: or

- **or:** `(x or y)` evaluates to `True` if and only if either `x` or `y` are `True` expressions
  - `a = ( (2 != 3) or ('yes' == 'yes')) → True`
  - `a = ( (2 != 2) or ('yes' == 'yes')) → True`
  - `a = ( (2 != 3) or ('yes' == 'no')) → True`
  - `a = ( (2 != 2) or ('yes' == 'no')) → False`
- is color either blue or red? check if the remainder of `x`, is 2 or 3 ... as long as one condition is satisfied, the expression evaluates to `True`
- `x` equal to 5,6, or 7:  
`(x == 5) or (x == 6) or (x == 7)`
  - ✓ **+, overloaded operator (but result changes type)**
  - ✓ Logical `or` is equivalent to addition

p	q	p∨q
T	T	T
T	F	T
F	T	T
F	F	F

Logical truth  
table for OR

A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

**OR**

`x = True`

`y = False`

`a = x + y`

`print(a, type(a)) → True <class 'int'>`

`print(y+y, type(y+y)) → False <class 'int'>`

`print(x+5, type(x+5)) → 6 <class 'int'>`

`print(y*5.6, type(y*5.6)) → 0.0 <class 'float'>`

# Boolean types and Logical operators: not

- **not**: (**not** x) evaluates to `True` if and only if x is a `False` expression
  - `a = not (2 != 3) → True`
  - `a = not ('yes' == 'yes') → False`
- Useful anytime the negation of an expression is needed

p	~p
T	F
F	T

A	not A
0	1
1	0

**NOT**

Logical truth  
table for NOT

`x = True`

`y = False`

`print(1-x, type(1-x)) → False <class 'int'>`

`print(1-y, type(1-y)) → True <class 'int'>`

- ✓ —, overloaded operator (but result changes type)
- ✓ Logical not is equivalent to one's complement

# Precedence rules among operators

Level	Category	Operators
7(high)	exponent	**
6	multiplication	*,/,//,%
5	addition	+, -
4	relational	==, !=, <=, >=, >, <
3	logical	not
2	logical	and
1(low)	logical	or

`x*5 >= 10 and y-6 <= 20`

First the arithmetic (`x*5`), then the relations (`x*5 > 5`, `y-6 <= 20`), and finally the logical and

# Flow control with conditional execution (branching)

1. Start with an arbitrary guess value,  $g$
2. **If**  $g \cdot g$  is close enough to  $x$  (with a given numeric approximation)  
**Then** Stop, and say that  $g$  is the answer
3. **Otherwise** create a new guess value by averaging  $g$  and  $x/g$ :

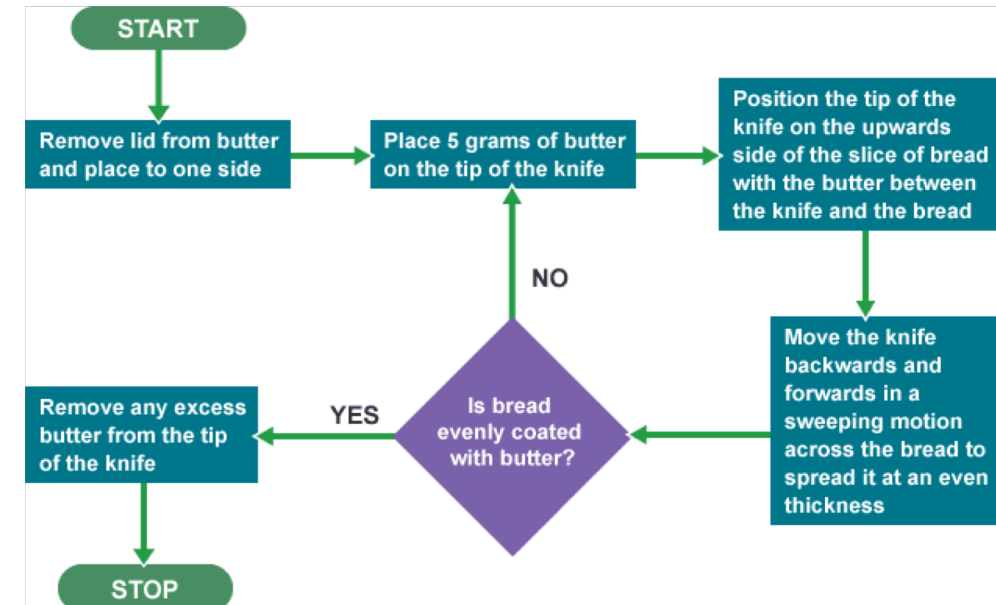
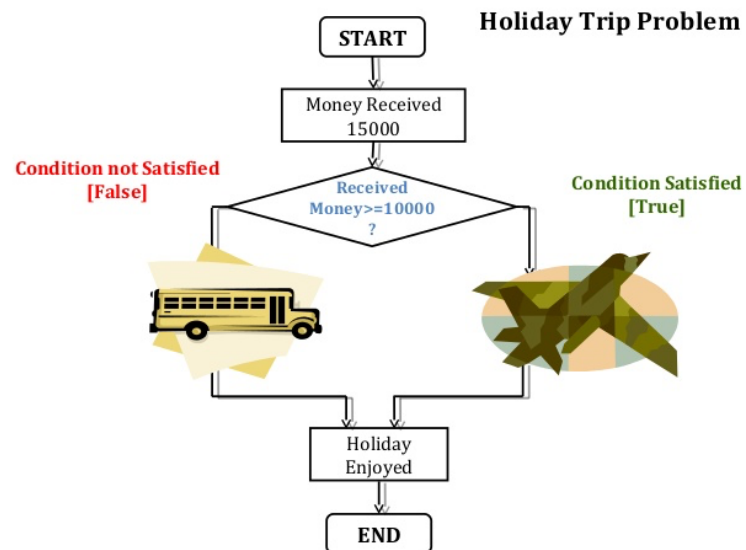
$$g = \frac{(g + x/g)}{2}$$

4. **Repeat** the steps 2 and 3 until  $g \cdot g$  is close enough to  $x$

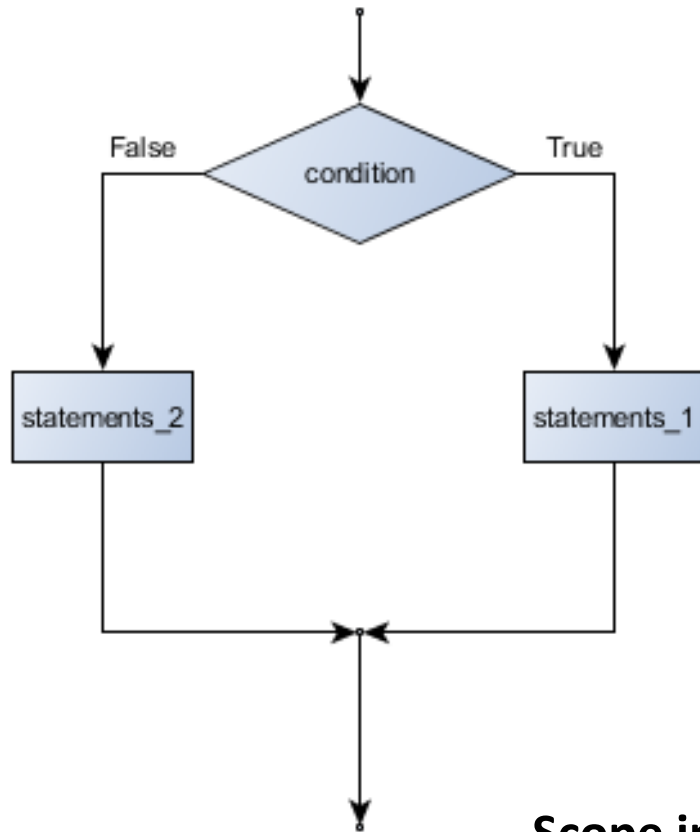
State of the program + input data



Branching ...



# Flow control with conditional execution: if-else

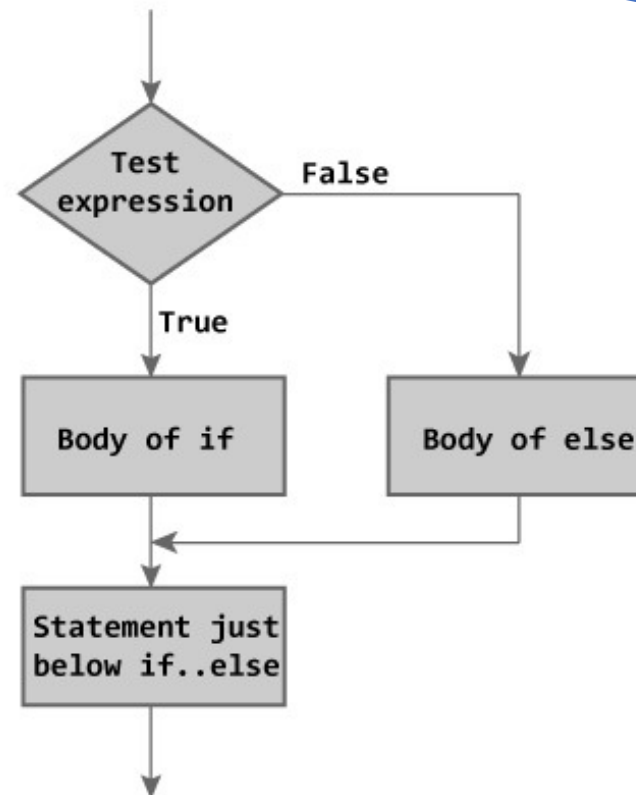


**Scope indentation**  
*Use TABS!*

```
if boolean_expression_is_true:
    do_something
else:
    do_something_else
keep_going_on_with the program
```

colons

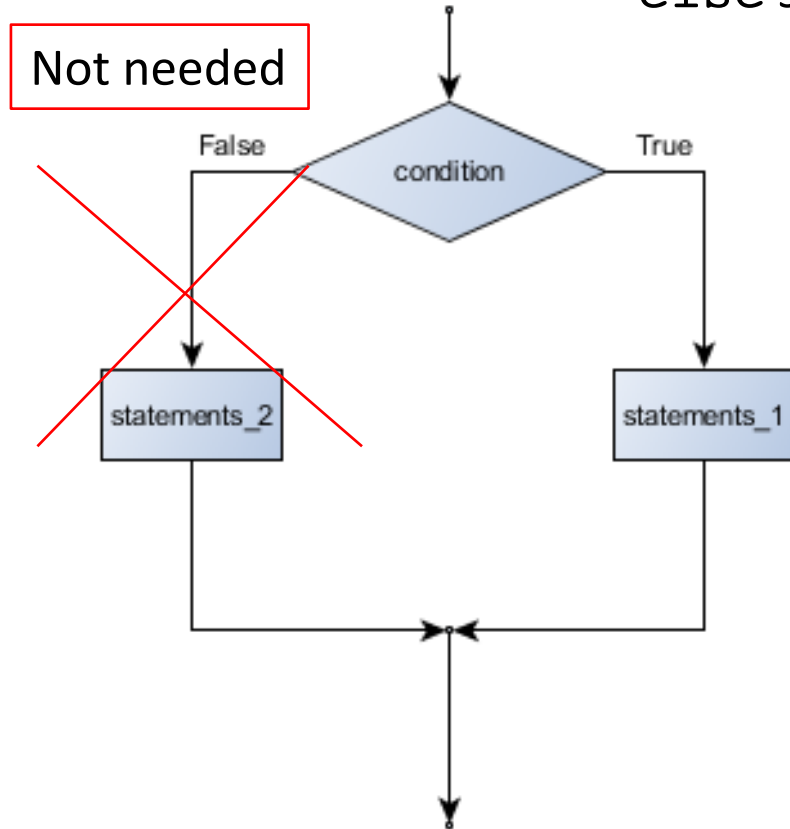
if-else keywords



```
if a > b:
    print(a)
else:
    print(b)
x = "whatever next"
print(x)
```

# Flow control with conditional execution: if

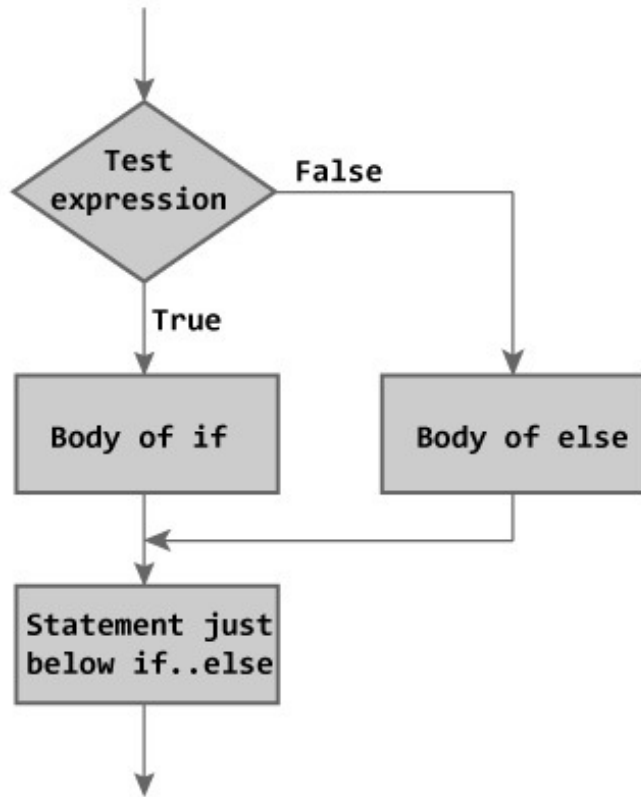
- Sometimes we only have one condition to check, one branch in the program flow
- `else` statement is *not* required



```
if boolean_expression_is_true:
    do_something
keep_going_on_with the program
```

```
if a > b:
    print(a)
x = "whatever next"
print(x)
```

# Flow control with conditional execution: scoping



```
if boolean_expression_is_true:  
    do_something
```

```
else:
```

```
    do_something_else  
keep_going_on_with the program
```

Local scopes

```
if a > b:  
    y = 3  
    print(a)  
else:  
    print(b, y)  
x = "whatever next"  
print(x, y)
```

It may be not defined

# Flow control with conditional execution: if-elif-else,



```
if boolean_expression_1_is_true:
    do_something_1
elif: boolean_expression_2_is_true:
    do_something_2
elif: boolean_expression_3_is_true:
    do_something_3
```

....

```
else:
    do_something_else
keep_going_on_with the program
```

➤ Also in this case, the else part is optional

```
if a > b:
    print(a)
elif a == b:
    a = a + 1
elif a == b - 1:
    print(b)
x = "whatever next"
```

```
if a > b:
    print(a)
elif a == b:
    a = a + 1
elif a == b - 1:
    print(b)
else:
    print(a + b)
x = "whatever next"
print(x)
```

