15-110 Principles of Computing – F19

Lecture 5:
Boolean types, Conditionals

Teacher:
Gianni A. Di Caro

So far about Python …

- Basic elements of a program:
  - Literal objects
  - Variables objects
  - Function objects
  - Commands
  - Expressions
  - Operators

- Object properties
  - Literal vs. Variable
  - Type
  - Scalar vs. Non-scalar
  - Immutable vs. Mutable

- Numeric data types:
  - `int`
  - `float`
- Logical data types:
  - `bool`

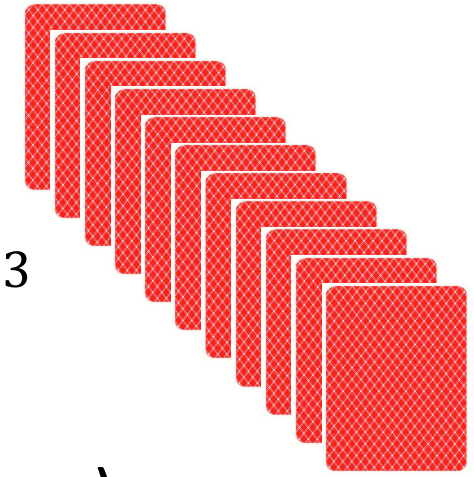
- Character data types:
  - `str`
- None data type:
  - `None`

- Utility functions (built-in):
  - `print(arg1, arg2, …)`
  - `type(obj)`
  - `id(obj)`
  - `int(obj)`
  - `float(obj)`
  - `bool(obj)`
  - `str(obj)`

- Operators:
  - `=`
  - `+`
  - `-`
  - `/`
  - `//`
  - `%`
  - `**`

What do we weed more?

- **Our NIM game:**
  - 11 items (e.g., matches, cards) are on the table
  - Two players taking turn
  - At each turn a player removes k items from the table, $1 \leq k \leq 3$
  - The player taking the last item(s) loses the game



Algorithm (winning strategy for first player):

Storing, accessing, updating values

1. *state* = 11
2. Player 1: Remove two items
3. *state* = $11 - 2$
4. Player 2: Remove k items
5. *state* = *state* $- k$
6. Player 1: Remove $4 - k$ items
7. *state* = *state* $-(4 - k)$
8. **Repea**t from 4 **until** there are no more items to remove

iteration

conditional

Boolean types and expressions

- `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
  - `(2+3) > 4` → `True`
  - `5 < x` → `True or False` depending on `x`
  - `2*x > y` → `True or False` depending on `x, y`

Comparison (Relational) operators

- 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,… → Relational operators are **overloaded operators**

Boolean types and Comparison (Relational) operators

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

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

```
n1 = 5.5
n2 = 7.1
d = n2 == n1
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))

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

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

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

16 different examples using relational operators

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

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

```
n1 = 5.5
n2 = 7.1
d = n2 > n1
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))

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

```
n1 = 5.5
n2 = 5.5
d = n2 >= n1
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 $0 \leq x \leq 5$?

```
in_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_ok = (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∧q
T	T	T
T	F	F
F	T	F
F	F	F

Logical truth table for AND

- 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  <class 'float'>
```

✓ *, overloaded operator (but result changes type)

✓ Logical `and` is equivalent to 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`

p	q	pvq
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

- 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)
```

```
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'>
```

✓ +, overloaded operator (but result changes type)

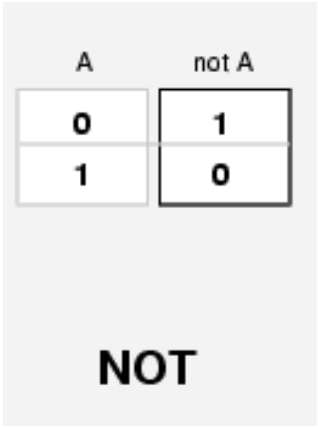
✓ Logical `or` is equivalent to addition

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)` → `False`
  - `a = not ('yes' == 'yes')` → `False`

- Useful anytime the *negation* of an expression is needed

p	~p
T	F
F	T



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`) and then (`y-6`), then the relations (`>= 10`, `<= 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)

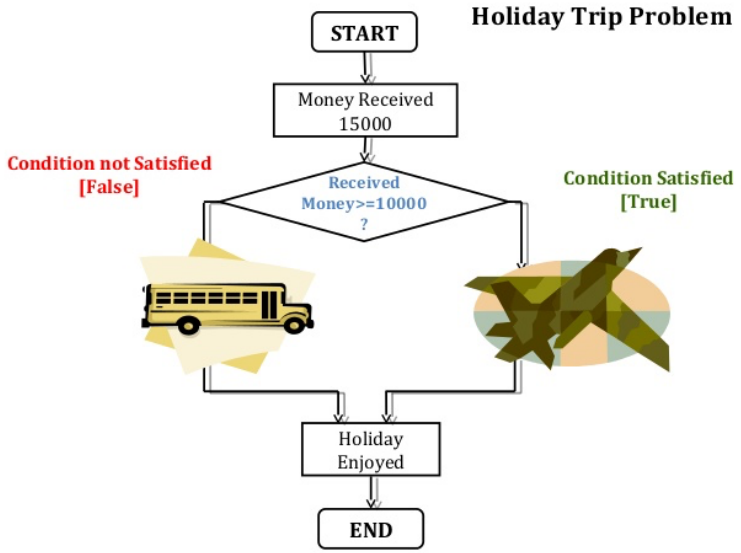
   Then `Stop`, and say that g is the `answer`
3. **Otherwise** create a new guess value by averaging g and x/g:

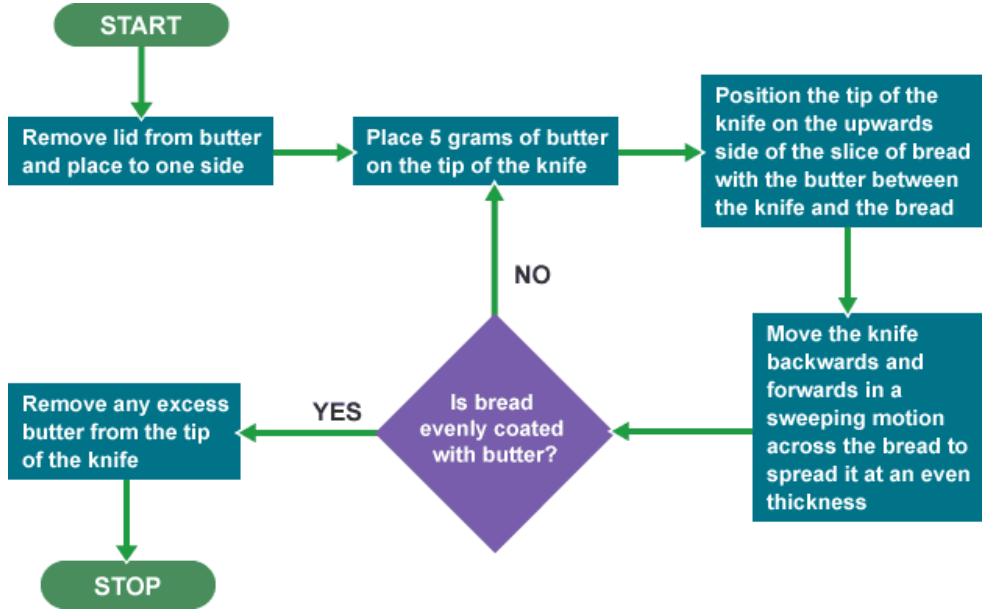
$$g = \frac{(g + x/g)}{2}$$

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

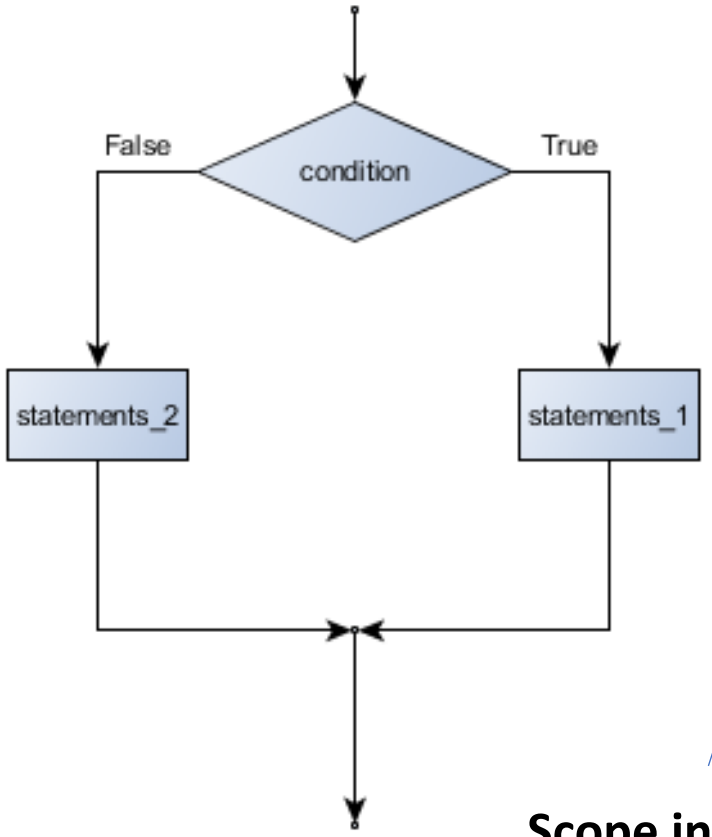
State of the program + input data

Branching …





Flow control with conditional execution: if-else

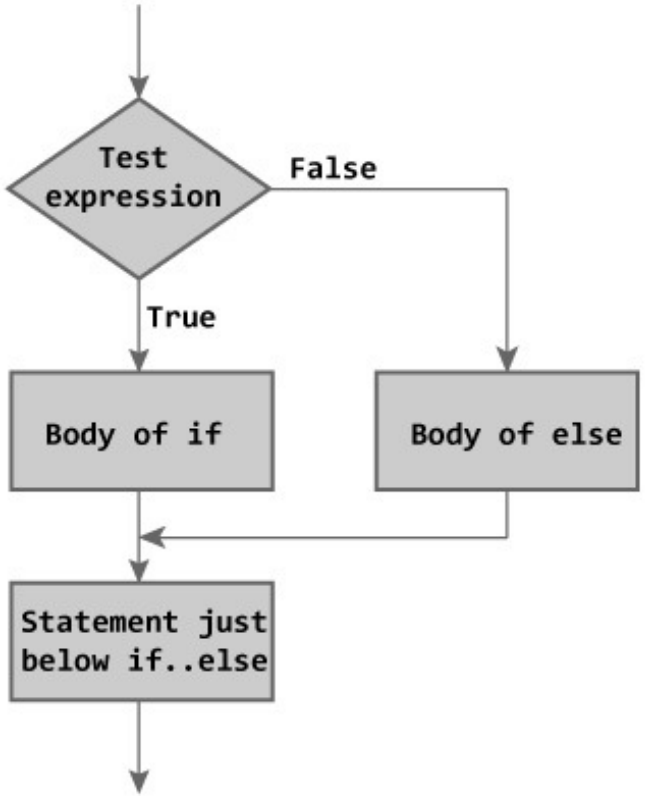



```
if boolean_expression_is_true:
    do_something
else:
    do_something_else
keep_going_on_with the program
```

colons

if-else keywords

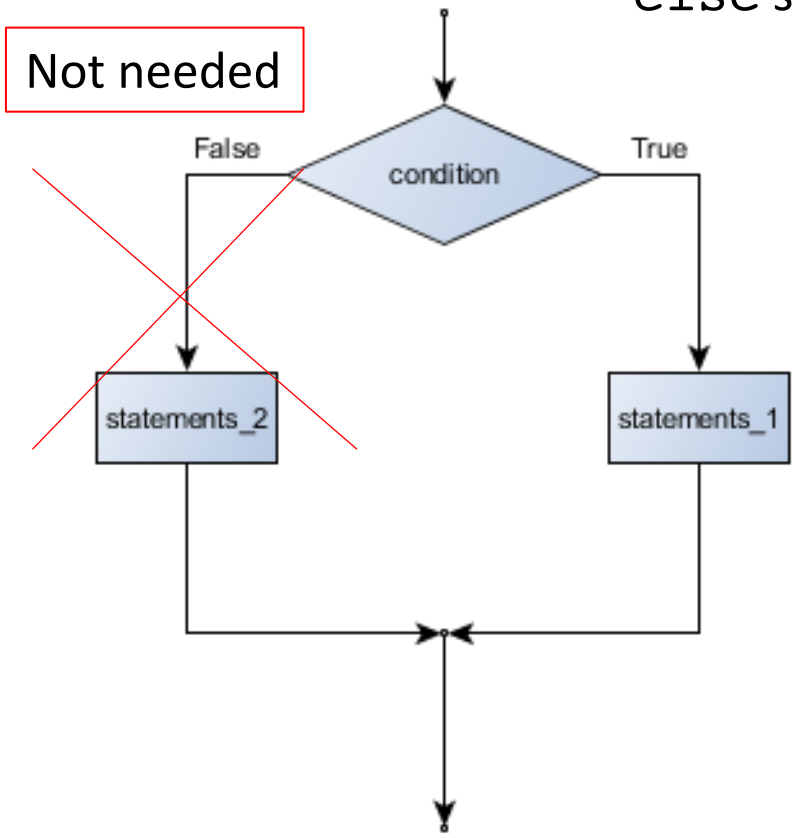
Scope indentation
Use TABS!




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

Flow control with conditional execution: if

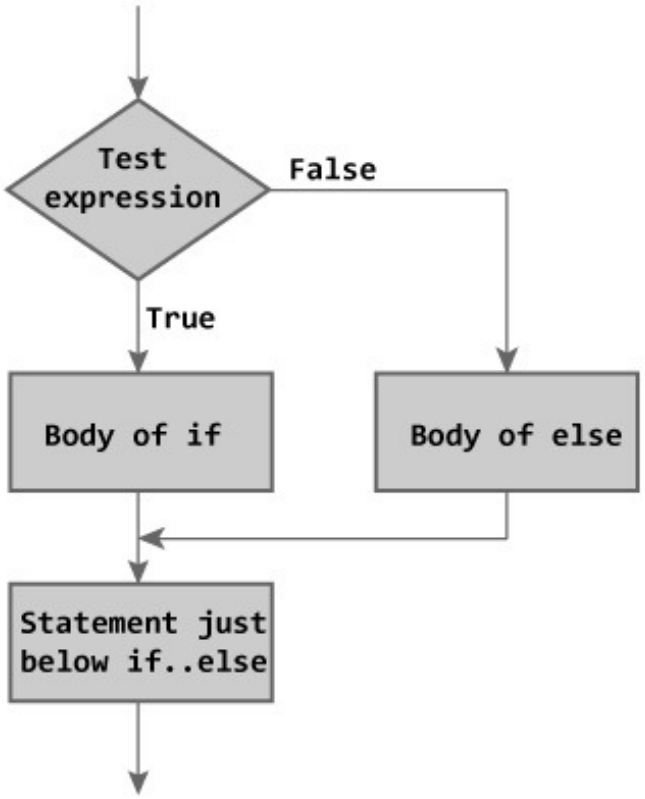
- Sometimes we only have one condition to check, one branch in the program flow
- `else` statement is *not* required




```
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



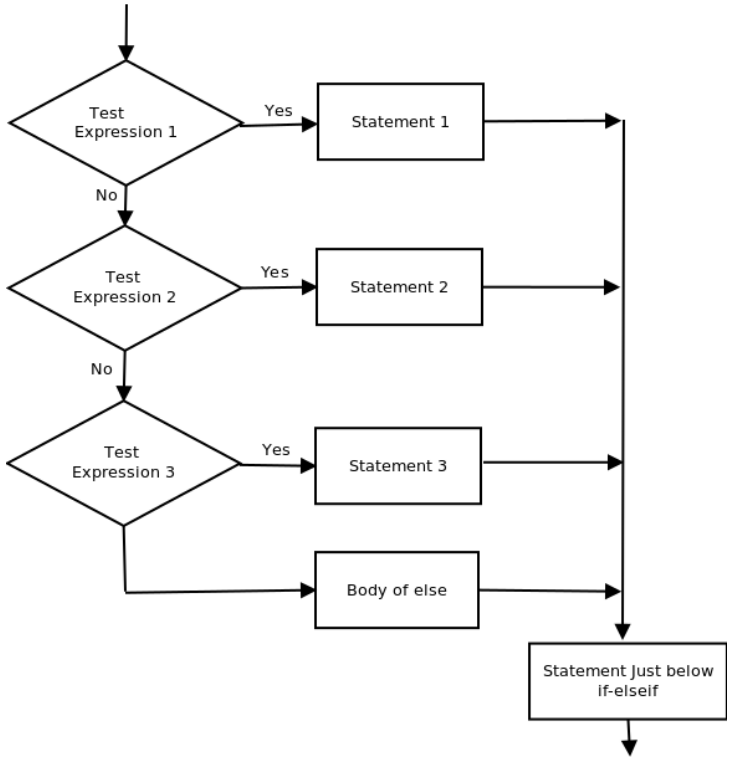

```
if boolean_expression_is_true:
    do_something
else:
    do_something_else
keep_going_on_with the program
```

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)
```