PYTHON GUIDE- PART 3

EQUALITY OPERATORS

Equality operators

An ***equality operator*** checks whether two operands' values are the same (==) or different (!=). Note that equality is ==, not just =.

An expression involving an equality operator evaluates to a Boolean value. A ***Boolean*** is a type that has just two values: True or False.

Table 4.1.1: Equality operators.

Equality operators	Description	Example (assume x is 3)
==	a **==** b means a is equal to b	x == 3 is true x == 4 is false
!=	a **!=** b means a is not equal to b	x != 3 is false x != 4 is true

RELATIONAL OPERATORS

Relational operators

A ***relational operator*** checks how one operand's value relates to another, like being greater than.

Some operators like >= involve two characters. A programmer cannot arbitrarily combine the >, =, and < symbols; only the shown two-character sequences represent valid operators.

Table 4.1.2: Relational operators.

Relational operators	Description	Example (assume x is 3)
<	a **<** b means a is less than b	x < 4 is true x < 3 is false
>	a **>** b means a is greater than b	x > 2 is true x > 3 is false
<=	a **<=** b means a is less than or equal to b	x <= 4 is true x <= 3 is true x <= 2 is false
>=	a **>=** b means a is greater than or equal to b	x >= 2 is true x >= 3 is true x >= 4 is false

COMPARING DIFFERENT DATATYPES

Comparing characters, strings, and floating-point types

The relational and equality operators work for integer, character, and floating-point built-in types.

Floating-point types should not be compared using the equality operators, due to the imprecise representation of floating-point numbers, as discussed in a later section.

The operators can also be used for the string type. Strings are equal if they have the same number of characters and corresponding characters are identical. If string my_str = 'Tuesday', then (my_str == 'Tuesday') is true, while (my_str == 'tuesday') is false because T differs from t.

The types of the values being compared determines the meaning of a comparison. If both values are numbers, then the numbers are compared arithmetically (`5 < 2` is False). Comparisons that make no sense, such as `1 < 'abc'` result in a TypeError.

Comparison of values with the same type, like `5 < 2`, or `'abc' >= 'ABCDEF'`, depend on the types being compared.

- Numbers are arithmetically compared.
- Strings are compared by converting each character to a number value (ASCII or Unicode), and then comparing each character in order. Most string comparisons use equality operators "==" or "!=", as in `today == 'Friday'.`
- Lists and tuples are compared via an ordered comparison of every element in the sequence. Every element between the sequences must compare as equal for an equality operator to evaluate to True. Relational operators like < or > can also be used: The order is determined by the first mismatching elements in the sequences. For example, if x = [1,2] and y = [1,3], then evaluating `x < y` first evaluates `1 < 1`, then `2 < 3`, which produces a value of True.
- Dictionaries are compared by sorting the keys and values of each dictionary and then comparing them as lists.

ERRORS & EXPRESSIONS

Common errors

A common error is to use = rather than == in an if-else expression, as in: `if numDogs = 9:`. In such cases, the interpreter should generate a syntax error.

Another common error is to use invalid character sequences like =>, !<, or <>, which are *not* valid operators.

4.2 Boolean operators and expressions

Booleans and Boolean operators

A ***Boolean*** refers to a value that is either True or False. Note that True and False are keywords in Python and must be capitalized. A programmer can assign a Boolean value by specifying True or False, or by evaluating an expression that yields a Boolean.

Figure 4.2.1: Creating a Boolean.

```
my_bool = True   # Assigns the boolean value True to my_bool

my_val  = 5
is_small = my_val < 3 # Evaluates the expression and assigns False to is_small
```

A ***Boolean operator*** treats operands as True or False and evaluates to a value of True or False. Boolean operators include `and`, `or`, and `not`. A ***Boolean expression*** is an expression using Boolean operators.

BOOLEAN OPERATORS

Table 4.2.1: Boolean operators.

Boolean operator	Description
a **and** b	***Boolean AND***: True when both operands are True.

a **or** b	***Boolean OR***: True when at least one operand is True.
not a	***Boolean NOT*** (opposite): True when the single operand is False (and False when operand is True).

Table 4.2.2: Boolean operators examples.

Given age = 19, days = 7, user_char = 'q'

`(age > 16) and (age < 25)`	True, because both operands are True.
`(age > 16) and (days > 10)`	False, because both operands are not True (days > 10 is False).
`(age > 16) or (days > 10)`	True, because at least one operand is True (age > 16 is True).
`not (days > 10)`	True, because operand is False.
`not (age > 16)`	False, because operand is True.
`not (user_char == 'q')`	False, because operand is True.

4.3 ORDER OF EVALUATION

Precedence rules

The order in which operators are evaluated in an expression are known as ***precedence rules***. Arithmetic, logical, and relational operators are evaluated in the order shown below.

Table 4.3.1: Precedence rules for arithmetic, logical, and relational operators.

Operator/Convention	Description	Explanation
()	Items within parentheses are evaluated first	In `(a * (b + c)) - d`, the + is evaluated first, then *, then -.
* / % + -	Arithmetic operators (using their precedence rules; see earlier section)	`z - 45 * y < 53` evaluates * first, then -, then <.
< <= > >= == !=	Relational, (in)equality, and membership operators	`x < 2 or x >= 10` is evaluated as `(x < 2) or (x >= 10)` because < and >= have precedence over or.
not	not (logical NOT)	`not x or y` is evaluated as `(not x) or y`
and	Logical AND	`x == 5 or y == 10 and z != 10` is evaluated as `(x == 5) or ((y == 10) and (z != 10))` because and has precedence over or.
or	Logical OR	`x == 7 or x < 2` is evaluated as `(x == 7) or (x < 2)` because < and == have precedence over or

Common error: Missing parentheses

A common error is to write an expression that is evaluated in a different order than expected. Good practice is to use parentheses in expressions to make the intended order of evaluation explicit. For example, a programmer might write:

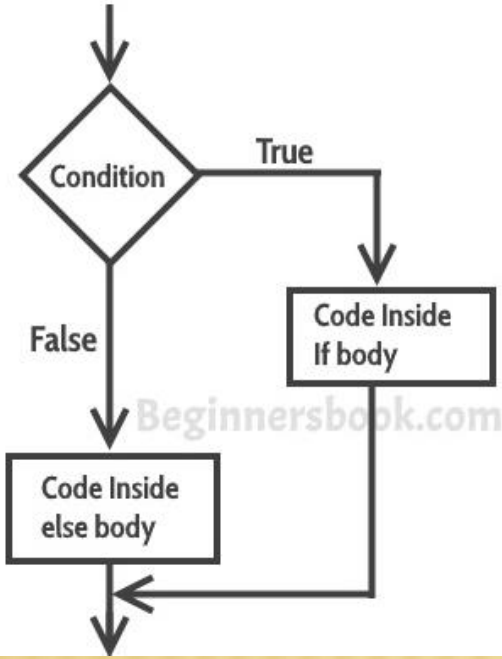
- `not a == b` intending to mean `(not a) == b`, but in fact the interpreter computes `not (a == b)` because equality operators (==) have precedence over logical operations (not).
- `w and x == y and z` intending `(w and x) == (y and z)`, but the interpreter computes `(w and (x == y)) and z` because == has precedence over and.
- `not x + y < 5` intending `(not x) + y < 5`, but the interpreter computes `not ((x + y) < 5)` because the addition operator + has the highest precedence and is computed first, followed by the relational operation <, and finally the logical not operation.

IF -> ELSE

If-else statement

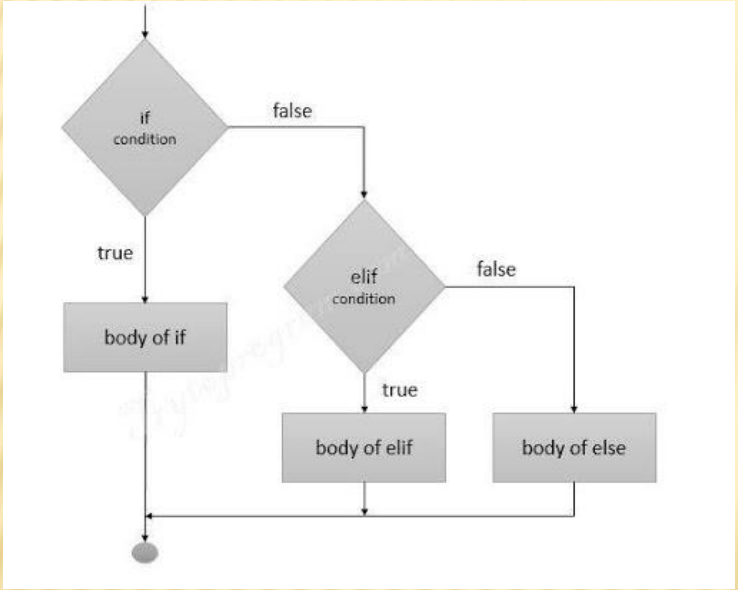
An ***if-else*** statement executes one group of statements when an expression is true and another group of statements when the expression is false.

Construct 4.5.1: If-else statement.




```
# Statements that execute before the branches

if expression:
   # Statements that execute when expression is true (first branch)
else:
   # Statements that execute when expression is false (second branch)

# Statements that execute after the branches
```

MULTIBRANCHES



Multi-branch if-else statements

An if-else statement can be extended to have three (or more) branches. Each branch's expression is checked in sequence. As soon as one branch's expression is found to be true, that branch's statement executes (and no subsequent branch is considered). If no expression is true, the else branch executes.

Construct 4.5.2: Multi-branch if-else statement. Only 1 branch will execute.


```
if expression1:
   # Statements that execute when expression1 is true
   # (first branch)
elif expression2:
   # Statements that execute when expression1 is false and expression2 is true
   # (second branch)
else:
   # Statements that execute when expression1 is false and expression2 is false
   # (third branch)
```


The ***equality operator ==*** evaluates to true if the left side and right side are equal. Ex: If num_years holds the value 10, then the expression `num_years == 10` evaluates to true.

Note that the equality operator is ==, not =.

Note: Good Practice is to be as explicit as possible and reduce the chances of logical errors by using the less-than-or-equal-to rather than just less-than. Thus, the constant is the number of relevance, e.g., less-than-or-equal-to 39 rather than less-than 40, because 40 is not the relevant number.

Figure 4.5.1: Multi-branch if-else example: Anniversaries.

```
num_years = int(input('Enter number years married: '))

if num_years == 1:
   print('Your first year -- great!')
elif num_years == 10:
   print('A whole decade -- impressive.')
elif num_years == 25:
   print('Your silver anniversary -- enjoy.')
elif num_years == 50:
   print('Your golden anniversary -- amazing.')
else:
   print('Nothing special.')
```

```
Enter number years married: 10
A whole decade -- impressive.

...

Enter number years married: 25
Your silver anniversary -- enjoy.

...

Enter number years married: 30
Nothing special.

...

Enter number years married: 1
Your first year -- great!
```

4.6 NESTED IF-ELSE

Nested if-else statements

A branch's statements can include any valid statements, including another if-else statement, which are known as ***nested if-else*** statements.

The below Python Tutor tool traces a Python program's execution. The Python Tutor tool is available at www.pythontutor.com.

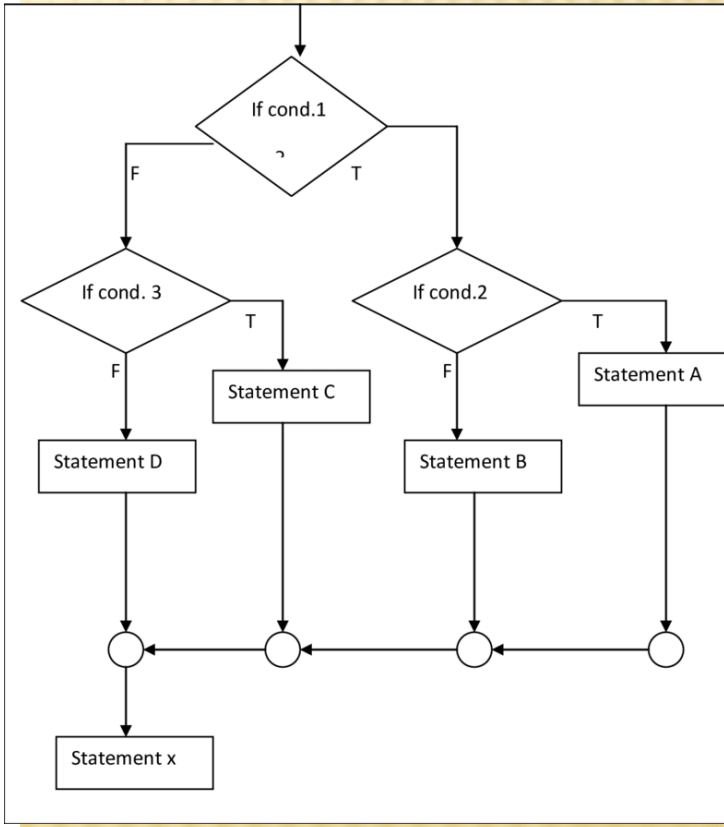
PARTICIPATION ACTIVITY 4.6.1: Nested if-else

```
1  user_choice = 2  # Hardcoded values for this tool. Could be input(
2  num_items = 5
3
4  if user_choice == 1:
5     print('user_choice is 1')
6  elif user_choice == 2:
7     if num_items < 0:
8        print('user_choice is 2 and num_items < 0')
9     else:
10       print('user_choice is 2 and num_items >= 0')
11 else:
12    print('user_choice is neither 1 or 2')
```

<< First < Back Step 1 of 6 Forward > Last >>

line that has just executed
next line to execute

Frames Objects



DISTINGUISHING THE STATEMENTS

Multiple distinct if statements

Sometimes the programmer has multiple if statements in sequence, which looks similar to a multi-branch if-else statement but has a very different meaning. Each if statement is independent, and thus more than one branch can execute, in contrast to the multi-branch if-else arrangement.

PARTICIPATION ACTIVITY 4.6.3: Multiple distinct if statements.

```
1  user_age = 26  # Hardcoded for this tool. Could replace with "int(ir
2
3  # Note that more than one "if" statement can execute
4  if user_age < 16:
5     print('Enjoy your early years.')
6
7  if user_age > 15:
8     print('You are old enough to drive.')
9
```



```
10 if user_age > 17:
11    print('You are old enough to vote.')
12
13 if user_age > 24:
14    print('Most car rental companies will rent to you.')
15
16 if user_age > 34:
```


<< First < Back Step 1 of 9 Forward > Last >>

line that has just executed
next line to execute

4.7 CODE BLOCKS & INDENTATION

Code blocks

A ***code block*** is a series of statements that are grouped together. A code block in Python is defined by its indentation level, i.e., the number of blank columns from the left edge. The initial code block is not indented. A new code block can follow a statement that ends with a colon, such as an "if" or "else". In addition, a new code block must be more indented than the previous code block. The program below includes comments indicating where each new code block begins.

The amount of indentation used to indicate a new code block can be arbitrary, as long as the programmer uses the same indentation consistently for each line in the block. Good practice is to use the standard recommended 4 columns per indentation level.

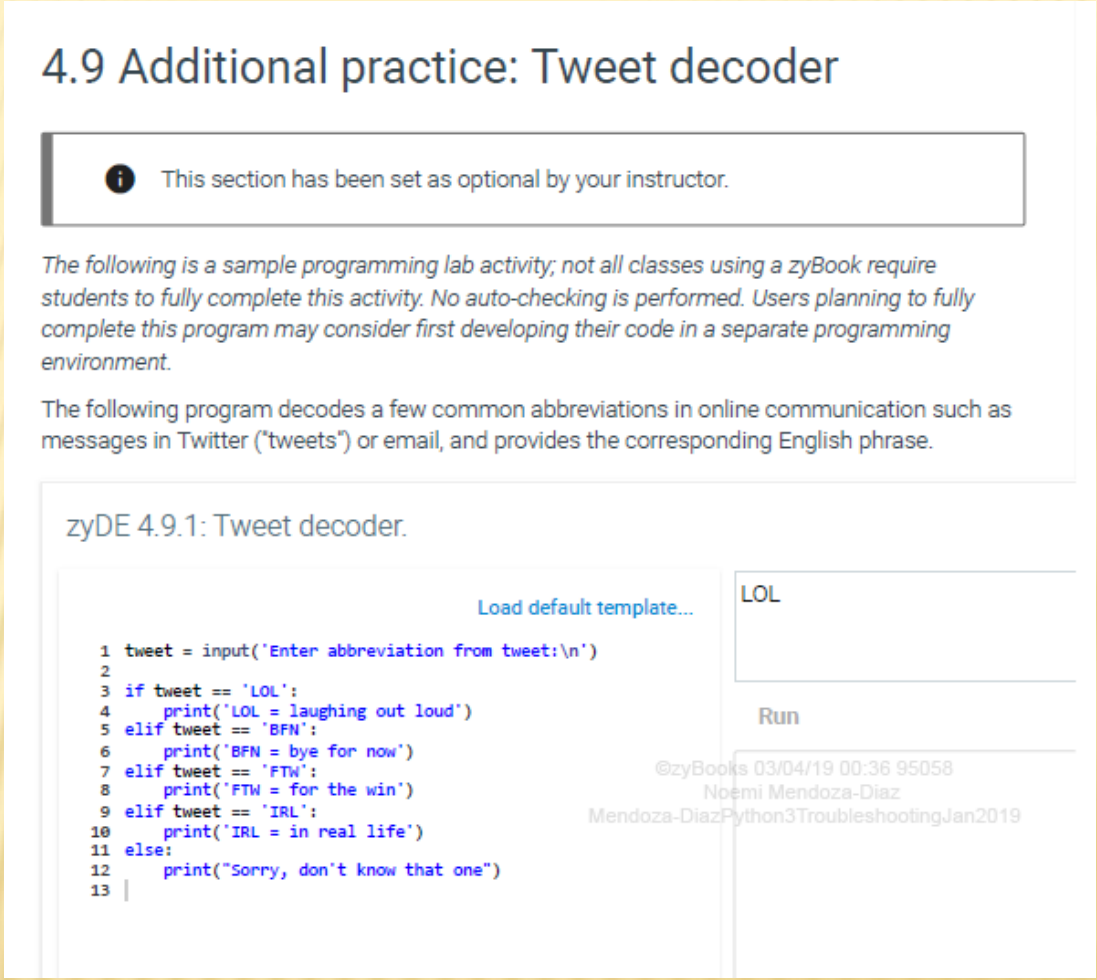
A common error for new Python programmers is the mixing of tabs and spaces. Never mix tabs and spaces for indentation in the same program. Many editors consider a tab to be equivalent to either 3 or 4 spaces, while in Python a tab is equivalent only to another tab. A program that mixes tabs and space to indent code blocks will automatically generate an IndentationError from the interpreter in Python 3. A good practice is to use spaces only when indenting code, and to set text editor options to automatically use spaces when possible.

Figure 4.7.1: Code blocks are indicated with indentation.


```
# First code block has no indentation

model = input('Enter car model: ')
year = int(input('Enter year of car manufacture: '))

antique = False
domestic = False

if year < 1970:
    # New code block has indentation of 4 columns
    antique = True

# Back to code block 0

if model in ['Ford', 'Chevrolet', 'Dodge']:
  # New code block has indentation of 2 columns
  # Any amount of indentation > 0 is OK.
  domestic = True

# Back to code block 0

if antique:
    # New code block has indentation of 4 columns
    if domestic:
        # New block has 4 additional columns (8 total)
        print('My own model-T still runs like a
charm...')
```

```
Enter car model: Ford
Enter year of car manufacture: 1918
My own model-T still runs like a
charm...
```

ADDITIONAL PRACTICE...



4.9 Additional practice: Tweet decoder

This section has been set as optional by your instructor.

The following is a sample programming lab activity; not all classes using a zyBook require students to fully complete this activity. No auto-checking is performed. Users planning to fully complete this program may consider first developing their code in a separate programming environment.

The following program decodes a few common abbreviations in online communication such as messages in Twitter ("tweets") or email, and provides the corresponding English phrase.

zyDE 4.9.1: Tweet decoder.

Load default template...

```
1  tweet = input('Enter abbreviation from tweet:\n')
2
3  if tweet == 'LOL':
4      print('LOL = laughing out loud')
5  elif tweet == 'BFN':
6      print('BFN = bye for now')
7  elif tweet == 'FTW':
8      print('FTW = for the win')
9  elif tweet == 'IRL':
10     print('IRL = in real life')
11 else:
12     print("Sorry, don't know that one")
13
```

LOL

Run