Unit 1.2: Conditional Branching

August 23, 2021

In the last unit we got started with Python. We learned about simple and formatted printouts, how to read simple user inputs, the basic data types, arithmetic expressions, and variables. You have used these in your first programming exercises.

In this unit we will talk about boolean expressions and conditional branching. Boolean expressions allow us to express conditions and to check at runtime if they are true or false. With the if statement we can deviate from the normal sequential control-flow depending on whether a certain condition is true or not. This allows us to let our program behave differently in different situations.

In the next unit we will deal with the implementation of repetitive behavior (loops).

Boolean Values

Booleans are data types that have only two possible values: `True` and `False` , or `1` and `0` . They represent the two truth values of Boolean logic and algebra (named after the English mathematician, philosopher and logician George Boole, who worked on this logic in the 19th century).

In Python we can assign the values `True` and `False` to variables just like strings and numbers, and also use and for example print them as usual:

In [1]:

```python
a = True
b = False
print(f"a is {a} and b is {b}.")
```

```
a is True and b is False.
```

Boolean Expressions

What can we do with Boolean values, and where do they actually come from?

Boolean algebra has three basic operations: `not` , `and` and `or` (evaluation in this order of precedence). The following Python program demonstrates how they work:

In [2]:

```python
a = True
b = False
print(f"a is {a} and b is {b}.")
print(f"not {a} is {not a} and not {b} is {not b}.")
print(f"{a} or {b} is {a or b}.")
print(f"{a} and {b} is {a and b}.")
```

```
a is True and b is False.
not True is False and not False is True.
True or False is True.
True and False is False.
```

That is, the `not` operation turns a Boolean value into the other. The result of an `or` operation with two Boolean values is `True` if at least one of them is `True`. The result of an `and` operation is `True` if both input values are `True`. Typically the functional values of Boolean expressions for all possible combinations of values of their functional arguments are surveyed in truth tables, like shown below:

a	b	not a	not b	a or b	a and b
True	True	False	False	True	True
True	False	False	True	True	False
False	True	True	False	True	False
False	False	True	True	False	False

Based on these operations, also more complex expressions can be formed, for example:

In [3]:

```python
a = True
b = False
print(f"{a} => {b} is {not a or b}.")
print(f"{a} xor {b} is {(a and not b) or (not a and b)}.")
```

```
True => False is False.
True xor False is True.
```

Note that unlike strings and numbers, Boolean values cannot be simply read as input from the user. (If a user types in `True` or `False` at an input prompt, this is initially a string. By convention, Python considers any non-empty string to be `True`, so a type cast to Boolean would lead to a `True` value in both cases.)

Usually Boolean values originate from comparisons or other tests. In Python the basic comparison operators returning `True` or `False` are:

```
<     (less than)
>     (greater than)
<=    (less than or equal to)
>=    (greater than or equal to)
==    (equal to)
!=    (not equal to)
```

Equality and non-equality can be checked for objects of all data types (also those that we have not yet discussed), while the less/greater than checks only work for data types for which an ordering relation is defined. This is for example the case for numbers (integers and floats) and strings, so they can easily be applied there:

In [4]:

```
a = 5.2
b = 3
print(f"{a} < {b} is {a < b}")
print(f"{a} <= {b} is {a <= b}")
print(f"{a} > {b} is {a > b}")
print(f"{a} >= {b} is {a >= b}")
print(f"{a} == {b} is {a == b}")
print(f"{a} != {b} is {a != b}")
```

```
5.2 < 3 is False
5.2 <= 3 is False
5.2 > 3 is True
5.2 >= 3 is True
5.2 == 3 is False
5.2 != 3 is True
```

In [5]:

```
a = "hello"
b = "world"
print(f"{a} < {b} is {a < b}")
print(f"{a} <= {b} is {a <= b}")
print(f"{a} > {b} is {a > b}")
print(f"{a} >= {b} is {a >= b}")
print(f"{a} == {b} is {a == b}")
print(f"{a} != {b} is {a != b}")
```

```
hello < world is True
hello <= world is True
hello > world is False
hello >= world is False
hello == world is False
hello != world is True
```

Note that comparisons can be chained arbitrarily, that is, and expression like `0 < x <= 10` is perfectly valid, but this is often not easy to read and understand. In most cases it is thus better coding style to use only pair-wise comparisons and chain them together with the Boolean `and` operator:

In [6]:

```
x = int(input("Enter an integer number: "))
is_in_range = 0 < x and x <= 10
print(f"Is {x} in range? {is_in_range}")
```

```
Enter an integer number: 12
Is 12 in range? False
```

This example also shows how comparisons that result in Boolean values and Boolean operations can be combined into more complex Boolean expressions. Here another, maybe more meaningful example:

In [7]:

```python
weekday = input("Which day of the week is it? ")
time = input("What time is it? (hh:mm) ")
is_lecturetime = (weekday == "Wednesday" or weekday == "Friday") \
and time >= "13:15" and time <= "15:00"
print(f"Lecture time? {is_lecturetime}")
```

```
Which day of the week is it? Wednesday
What time is it? (hh:mm) 12:45
Lecture time? False
```

Note that Python performs a "lazy evaluation" of Boolean expressions, that is, it will process the expression from left to right, but only until the point where a certain decision can be made if it evaluates to `True` or `False` . For example, when we enter “Tuesday” in the program above, the evaluation will stop after the check for Friday, at it is clear at this point that the and expression cannot become `True` any more. If we enter “Wednesday”, it will skip the test for Friday, but it still it needs to confirm both of the times to be sure that it is lecture time (and can stop after one of them fails). Taking into account this lazy evaluation behavior can help to write more efficient evaluations, for example by checking those things first that can already determine the result of the complete expression, or by performing "expensive" calls (e.g. to external databases) only when it is really needed for determining the outcome. In the program above it might be somewhat faster to check the times first and then the day, but for humans it is usually more logical to check the day first.

The `if` statement

Now let’s see how we can use Boolean expressions for implementing conditional branching in Python programs. This will allow us to not only print out the bare Boolean values as we did above, but to actually react on whether something is true or false. What we need is the `if` statement. Here is a simple example:

In [8]:

```python
name = input("What is your name? ")
if name == "Jane":
    print("Hello Jane, nice to see you again!")
```

```
What is your name? Anna
```

In the first line the program reads an input (name) from the user. In the next line, it checks if the name is “Jane”, and only if that is `True` the following line is executed. If it is `False` , nothing else happens.

If we want something to happen also for the case the checked condition is False, we can add an else statement. For example:

In [9]:

```python
name = input("What is your name? ")
if name == "Jane":
    print("Hello Jane, nice to see you again!")
else:
    print(f"Hello {name}, nice to meet you!")
```

```
What is your name? Anna
Hello Anna, nice to meet you!
```

And if we want to distinguish even more than two cases, we can furthermore add `elif` (Python’s abbreviation for “else if”) statements. For example:

In [10]:

```
name = input("What is your name? ")
if name == "Jane":
    print("Hello Jane, nice to see you again!")
elif name == "Steven":
    print("Hello Steven, nice to see you again!")
elif name == "Kim":
    print("Hello Kim, nice to see you again!")
else:
    print(f"Hello {name}, nice to meet you!")
```

```
What is your name? Kim
Hello Kim, nice to see you again!
```

With the if statement we can also rewrite the Boolean expression examples from above to produce nicer outputs:

In [11]:

```
x = int(input("Enter an integer number: "))
if 0 < x and x <= 10:
    print(f"{x} is in range.")
else:
    print(f"{x} is not in range")
```

```
Enter an integer number: 8
8 is in range.
```

In [12]:

```
weekday = input("Which day of the week is it? ")
time = input("What time is it? (hh:mm) ")
if (weekday == "Wednesday" or weekday == "Friday") and \
    time >= "13:15" and time <= "15:00":
    print("It is lecture time!")
elif (weekday == "Wednesday" or weekday == "Friday") and \
    time >= "15:15" and time <= "17:00":
    print("It is lab time!")
else:
    print("It is homework time!")
```

```
Which day of the week is it? Wednesday
What time is it? (hh:mm) 12:45
It is homework time!
```

So, the basic structure for conditional branching with in Python can be described as follows (where `elif` and `else` are optional and there can be more than one `elif`):

```
if <some condition>:
    <do something>
elif <some condition>:
    <do something else>
else:
    <do something else>
```

Two things are important to note here:

1. There can be more than one line of code after the `if/elif/else` statements that is executed for the respective case. These lines of code together are then called a block. Blocks can contain further `if/elif/else` statements.

2. Indentation is important. The `if/elif/else` blocks are indented by exactly four spaces (in most Python editors the TAB key will insert them automatically) to denote that they belong to the block for the respective case. The next code after the if-statement is simply not indented. (We will see later that Python uses this “meaningful whitespace” also for other constructs. Many other languages use additional parentheses for the same purpose.)

Here is a slightly more complex example to illustrate the above:

In [13]:

```python
a = float(input("Enter a number:"))
b = float(input("Enter another number:"))

if a == b:
    print("The numbers are the same")
    print("(What a boring input…)")
else:
    if a > b:
        print(f"{a} is greater than {b}.")
        print(f"{a/b:.1f} times greater, to be more precise.")
    else:
        print(f"{b} is greater than {a}.")
        print(f"{b/a:.1f} times greater, to be more precise.")

print("Thanks for using this program.")
```

```
Enter a number:12
Enter another number:12
The numbers are the same
(What a boring input…)
Thanks for using this program.
```

Exercises

1. Age Check

Write a program that asks the user to enter their age. If it is under 18, the program should display a message to refuse entry, otherwise invite the user to come in. The output should be something like:

```
How old are you (in years)? 15
Sorry, you are not allowed to enter.
```

```
How old are you (in years)? 29
Welcome! Please come in.
```

2. Divisibility

Write a program that asks the user to enter two integer numbers, then checks if the first number is divisible by the second number and informs the user accordingly. The output should be something like:

```
Please enter an integer number: 15
Please enter another integer number: 5
15 is divisible by 5.
```

```
Please enter an integer number: 12
Please enter another integer number: 5
12 is not divisible by 5.
```

3. BMI Calculation Revisited

Extend the BMI calculation program from the last homework so that after informing the user about the calculated BMI, it also prints out if the BMI is within the range that is generally considered normal (between 18.5 and 25) or higher (above 25) or lower (below 18.5) than that. The output of the modified program should then be something like:

```
Welcome to the BMI calculator.
What is your name? John Doe
What is your weight (in kg)? 78
What is your height (in m)? 1.82
Hello John Doe, your BMI is 23.5.
Your BMI is normal.
```

4. Extra: Diagnostics

Influenza-like illness is characterized by the patients having fever, cough, and one or more of the following symptoms: sore throat, joint and muscle pain, complete exhaustion. Write a Python program that asks a patient about the symptoms (if the patient has them or not), and from that determines if influenza-like illness is likely or not. Try to use only **one** Boolean expression for that. The output should be something like:

```
Do you have fever? (y/n) y
Do you have a cough? (y/n) y
Do you have a sore throat? (y/n) n
Do you have joint and muscle pain? (y/n) n
Do you experience complete exhaustion? (y/n) y
I think you have influenza-like illness. Please consult your GP.
Do you have fever? (y/n) y
Do you have a cough? (y/n) n
Do you have a sore throat? (y/n) y
Do you have joint and muscle pain? (y/n) y
Do you experience complete exhaustion? (y/n) n
I don't know what you have, but it does not seem to be influenza-like illne
ss. Please consult your GP.
```

In [ ]:

```

```