PROGRAMMING IN PYTHON I

Conditions and Loops

Andreas Schörgenhumer

Institute for Machine Learning

JOHANNES KEPLER
UNIVERSITY LINZ

Institute for
Machine Learning

Copyright Statement

Contact

Andreas Schörgenhumer

Institute for Machine Learning
Johannes Kepler University
Altenberger Str. 69
A-4040 Linz

E-Mail: schoergenhumer@ml.jku.at
Write mails only for personal questions
Institute ML Homepage

Motivation

- To solve tasks, we often have to make decisions, e.g.:
  - *Should I order food, yes or no?*
  - *Is x smaller than 10?*
  - *Does my list contain any items?*
- We want to incorporate such decisions via **conditions** that can then be either true or false
- Using such conditions, we can control the flow in our program:
  - Using **branches** to take a specific path
  - Using **loops** to repeat certain parts
- More details on control flow tools in Python: `https://docs.python.org/3/tutorial/controlflow.html`

Evaluating Conditions

- We already heard about the **boolean** data type
  - Can be True or False
- To make decisions, we need an **expression** that results in a boolean value (= boolean expression)
  - E.g.: *Yes or no?*, *Is 'a' equal to 'b'?*, *Is the value within a range?*, *Is x smaller than 10?*, ...
- **Comparisons**[1] are often used as such expressions, e.g.:

```
4 > 5  # -> False
3 < 4 < 5  # -> True
x == 100  # -> True if x equals 100
```

- **Logical operations** are used to modify and combine boolean expressions

[1] https://docs.python.org/3/library/stdtypes.html#comparisons

Logical Operations

- Also called boolean operations: **not**, **or**, **and**
- Truth tables with boolean `x` and boolean `y` (with F = False and T = True):

x	not x
F	T
T	F

x	y	x or y
F	F	F
F	T	T
T	F	T
T	T	T

x	y	x and y
F	F	F
F	T	F
T	F	F
T	T	T

Short-Circuit Evaluation

- For **or**, if the first boolean expression is already True, the evaluation of the second is skipped
- For **and**, if the first boolean expression is already False, the evaluation of the second is skipped
- Useful for safely checking potentially problematic conditions, e.g.:

```
a == 0 or b / a > 10
len(my_string) >= 1 and my_string[0] == "f"
```

Truth Value Testing

- There are multiple objects in Python that are interpreted as False when evaluated in a boolean context:
  - The special value **None**
  - The value 0 of all numeric data types
  - Empty sequences like strings and data structures
- All other objects are interpreted as True by default[2]
- This allows for shorter conditions, e.g., writing `my_string` instead of `len(my_string) >= 1` as part of a boolean expression

[2] Unless the object has a `__bool__()` method that evaluates to False or a `__len__()` method that returns zero.

Assertions

- Assertions are convenient to check if your code performs as expected, which is useful for debugging
- The **assert** statement allows to quickly implement sanity checks based on a boolean expression. They will raise an error if the condition evaluates to False. Example:

```
assert x > 0  # If x <= 0, aborts the program
assert x > 0, f"x was {x}"  # Same with message
```

- Assertions can be disabled when running Python with the `-O` option (`python -O ...`), which results in no additional run-time costs. This also means that you should not use assertions for general error/exception handling (since users can disable them) but for sanity checks only, i.e., to see whether your code behaves as expected.

Branching: if, elif, else

- Depending on some condition, we may want/not want to execute different parts of our code
- **If**, **elif**, **else** statements allow us to implement such a decision making

  If (=**if**) `cond1` is True, then do `code1`,
  otherwise if (=**elif**) `cond2` is True, then do `code2`,
  otherwise if (=**elif**) `cond3` is True, then do `code3`,
  otherwise (=**else**) do `code4`

- Evaluation is done from **top to bottom**
- Only **one branch** is ever **executed** (if there is no `else`, no branch might be executed at all)

Code Example

```
if x == 0:
    print("x was zero")
elif x > 0 and x <= 9:
    print("x was a single digit larger than zero")
else:
    y = x // 10
    print(y)
print("branching done")
```

- Note: Code within branches is assigned via **indentation**, i.e., no braces such as in other languages (e.g., Java)
- This holds for all code blocks in Python (loops, functions, classes, ...)
- Typically, 4 spaces

Pattern Matching

- Python 3.10 introduced a new feature called **structural pattern matching**, which can be constructed with the **match** statement. Simple example:

```
status = ... # Get HTTP integer status code
match status:
    case 400:
        print("Bad request")
    case 404:
        print("Not found")
    case 418:
        print("I'm a teapot")
    case _:
        print("Something's wrong with the Internet")
```

- This construct is actually much more powerful, however, we will not go into further details (see, e.g., https://peps.python.org/pep-0636/ for more details)

Loops: while, for

- Depending on some condition, we may want/not want to execute different parts of our code
- Sometimes, this also includes repeating the execution of code that was already executed
- **Loops** (**while** and **for**) allow us to implement such a repetition (this avoids code duplication)
- Loops might be executed 0 or more times (even infinitely often), depending on the loop's condition

While Loop

- The **while** loop in Python will repeat a part of code as long as the boolean expression of the loop is True (the loop condition), e.g.:
  - Ask user for password until they enter the correct password
  - Run some main routine of a micro controller until power is gone (e.g., keep driving around a small robot)
  - Keep optimizing network parameters until the output is close enough to the target
- Danger: This can (and often does) lead to endless/infinite loops if the expression is never False!

Code Example

```
x = 0
while x <= 0:
    x = int(input("Enter integer number > 0: "))
print(x)
```

For Loop

- The **for** loop in Python will repeat a part of code for each element in a so-called iterable of elements (strings, data structures, ...), e.g.:
  - For each character in the string `my_string`, compute the uppercase letter
  - For a given number of updates `range(n_updates)`, update the weights of a neural network

Code Example

```
my_string = "hello"
for char in my_string:
    up = char.upper()
    print(f"uppercase letter = {up}")
print("converted all letters to uppercase")
```

Manual Loop Control

- Within a loop, you can manually force to exit the loop or directly continue with the next loop iteration:
  - Keyword **break** to exit
  - Keyword **continue** to jump to the loop condition again
- Code below these keywords is ignored
- These keywords break the normal control flow of your program (sudden jumps in code), so use sparsely!