CHAPTER 2

Data Types and Variables

Lernziele

KdP2 Du implementierst gut strukturierte *Python-Programme* ausgehend von einer verbalen oder formalen Beschreibung unter adäquater Nutzung *imperativer Programmierkonzepte*.

KdP5 Du vergleichst die unterschiedlichen *Ausprägungen* der Programmierkonzepte- und paradigmen.

Organisatorisches

- Videolänge: ∼112 Min.
- Tutorien: 30.+31. Oktober
- Großplenum: 1. November, 8–10 Uhr im Hörsaal 1B
- Lernzentrum: 30. Oktober–3. November
- Großtutorium: 3. November, 8–10 Uhr im Hörsaal 1B
- Abgabe der Übungsaufgaben: 3. November, 14 Uhr

Contents

Übungsaufgaben

Hinweise: Aufgabe 3 ist freiwillig und wird nicht bewertet. Von Aufgabe 1 und 2 wird eine Aufgabe ausführlich korrigiert und 10 Punkte vergeben, eine Aufgabe wird nur abgehakt und 1 Punkt vergeben. Welche der beiden Aufgaben das jeweils sein wird, wird nach dem Abgabedatum entschieden.
Plagiate werden beim 1. Mal mit 0 Punkten auf dem gesamten Zettel und beim 2. Mal mit dem Verlust der aktiven Teilnahme geahndet. Bei jeder Programmieraufgabe wird erwartet, dass der Quelltext ausreichend kommentiert ist und getestet wurde.

1. **Zahlprobleme und Stringprobleme** (10 Punkte)

   a) Ordnet die einzelnen Übungsaufgaben den passenden Lernzielen zu.

   b) Beschreibe die einzelnen Probleme, die beim Arbeiten mit Zahlen in Programmiersprachen auftreten können. Welche dieser Probleme gibt es in Python und welche nicht? Begründung! Schreibe eigene Programme welche die in Python auftauchenden Probleme veranschaulichen – nutze in diesen Programmen Schleifen. Veranschauliche das Problem des Überlaufs mit Hilfe des Zweierkomplements anhand eigens gewählter Beispiele.

   c) Implementiere ein Programm, das einen String als Eingabe erwartet und das zugehörige Histogramm auf dem Bildschirm ausgibt. Das Histogramm eines Textes ist eine Tabelle (Dictionary), in der die absoluten Häufigkeiten der einzelnen Buchstaben bzgl. des Textes stehen. Benutze Strings und Dictionaries. (Bemerkung: Der String kann Buchstaben, Zahlen aber auch Sonderzeichen enthalten. Klein- und Großbuchstaben sind unterschiedliche Symbole.)

      Beispiel: Bei Eingabe des Wortes `HALLO` könnte das Programm folgende Ausgabe erzeugen: `{'H':1, 'A':1, 'L':2, 'O':1}`.

   d) In DNA-Strings sind die Symbole `A` und `T` bzw. `C` und `G` komplementär zueinander. Das *Rückwärtskomplement* eines DNA-Strings entsteht, indem zuerst die Zeichenkette umgedreht wird und anschließend alle Symbole durch ihr jeweiliges Komplement vertauscht werden. Beispiel: `ACGAATCG`, rückwärts geschrieben: `GCTAAGCA`, nun die Komplemente: `CGATTCGT`. Implementiere ein Programm, das einen DNA-String erwartet und das Rückwärtskomplement auf dem Bildschirm ausgibt. Vordefinierte Funktionen der Python-Bibliothek, die die Arbeit sehr viel einfacher machen könnten (z.B. die existierende reverse-Funktion), sind natürlich nicht erlaubt.

2. **Mastermind** (10 Punkte)

   In dieser Aufgabe sollt ihr schrittweise das Spiel Mastermind implementieren – statt Farbsequenzen werden wir jedoch DNA-Sequenzen nehmen.

   a) Implementiere ein Programm, welches eine zufällige DNA-Sequenz der Länge 6 auf dem Bildschirm anzeigt.

b) Implementiere ein Programm, welches zwei DNA-Sequenzen der Länge 6 als Eingabe erwartet und auf dem Bildschirm die Anzahl der Positionen ausgibt, an denen beide DNA-Sequenzen gleich sind.

Beispiel: Bei Eingabe von `ACAGGT` und `ATACGC` lautet die Antwort 3, weil beide Strings an den Stellen 0, 2 und 4 übereinstimmen.

c) Implementiere ein Programm, welches zwei DNA-Sequenzen der Länge 6 als Eingabe erwartet und auf dem Bildschirm die Anzahl gleicher Häufigkeiten der 4 Symbole ausgibt.

Beispiele: Bei Eingabe von `ACAGGT` und `ATACGC` ist die Antwort 2, weil die Anzahl der `A`'s und die Anzahl der `T`'s in beiden Strings gleich sind. Bei Eingabe von `ACATGC` und `ATACGC` ist die Antwort 4.

d) Implementiere ein Programm, welches eine zufällige DNA-Sequenz der Länge 6 erstellt und vom Benutzer/von der Benutzerin erraten werden muss: In jeder Runde wird eine 6-stellige DNA-Sequenz eingegeben und die Anzahl der korrekten Positionen [vgl. b)] sowie die Anzahl gleicher Häufigkeiten [vgl. c)] genannt. Das Programm terminiert, wenn die Sequenz richtig erraten wurde.

3. Sammeln (ohne Punkte)

a) Untersuche experimentell, wie oft man eine zufällige Zahl zwischen 1 und n wählen muss, bis jede Zahl vorgekommen ist. Mache für $n = 10$, 100 und 1000 jeweils 100 Experimeente und bestimme für jedes n das Minimum, das Maximum und den Mittelwert der Anzahl der Zahlen. Ein Experiment besteht dabei aus der notwendigen Anzahl von Versuchen, bis man alle Zahlen beobachtet hat.

(Bemerkung: Der Erwartungswert für die Anzahl der Versuche ist übrigens $\sum_{i=1}^{n} \frac{n}{i}$)

b) Erweitere dein Experiment aus a), sodass auch ermittelt wird, wie oft die häufigste Zahl vorgekommen ist. Mache eine analoge Statistik wie bei Aufgabe a).

2.1 The five facets of a variable

As a reminder, a variable in Python is a name that is associated with an object in memory. Moreover, over its lifetime, a variable can be associated with different objects of different types (Python is dynamically typed).

In most imperative programming languages a variable is characterized by at least five different properties – the facets of a variable.

2.1.1 Content

The content is the specific object with which the variable is associated, i.e., to be more precise the content is a binary sequence (representing the object). For example the content of `x = 42` would be the binary sequence `101010`.

2.1.2 Address

The address is the location of the object. To be more precise, if the object consists of more than one byte than the address of the object is the address of the first byte. In Python (and Scala) it is not possible to get the address of an object. However, languages that are close to hardware are able to show the address of variables and objects. These languages are for example C, C++, or Rust.

Nevertheless, in Python every object in the memory has a unique identifier which can be shown by `id(x)`. In many cases this unique identifier corresponds to the objects' address, but this is not guaranteed.

2.1.3 Size

Every object has a size – the number of bytes that are used to describe the object. In Python (and Scala) it is very difficult to get the exact size of an object. Hence, we will ignore this during the course. However, the languages that are close to hardware can easily show the size of variables and objects.

2.1.4 Data type

A data type is a collection of *values of the same kind*. The data type of an object tells us what *operations* are possible to apply on these values. Whenever a variable in Python is associated with an object, the type of this variable corresponds to the type of the associated object. In Python, it is possible to get the type of an object/variable by typing `type(x)`. We will take a closer look at the different types in a moment.

Attention: Since Python is a dynamically typed programming language the data type and the address of a variable can change over time. The following example demonstrates this behaviour:

```
>>> a = 0
>>> type(a)
<class int>
>>> a = True
>>> type(a)
<class bool>
```

In statically typed languages like Java, Scala, C++, C this is not the case.

2.1.5 Scope

The scope of a variable is the range of the program, in which the variable can be used. It starts with the *declaration* – the first appearance of an assignment like `x = expression`. After this assignment, the variable `x` can be used in other expressions within in the program. Later on, when introducing functions and methods we will distinguish between *global* and *local* variables.

Attention: Since Python uses an *interpreter* and not a *compiler* the following program will start but might produce a runtime error.

```
n = int(input("Gib eine Zahl ein: "))
if n <= 10:
    x = 12
print(x)
```

Whenever the user inputs a number larger than 10, the variable `x` is not declared and hence, `print(x)` causes a `NameError`.

Selbsttest: Schaut euch das Video 012 an. Welche der 5 Eigenschaften einer Variable können sich im Laufe ihrer Lebenszeit ändern und welche nicht?

2.2 Primitive data types

Every programming language has its own collection of types. The *type system* is one of the distinctive features of a programming language. We will now talk about the possible types in Python.

2.2.1 NoneType

The data type `NoneType` has only one possible value which is called `None`. This value is used to represent the *absence* of a result or value. Other languages have similar concepts, but the value is called different like `nil` or `null`. There are no operations for this data type.

```
>>> x = None
>>> print(x)    # nothing is printed on the screen
>>>
```

2.2.2 Booleans

The data type `bool` represents truth values in Boolean logic. There are two possible values: `True` and `False`. Operations include `and` (`True` if and only if both operands are `True`), `or` (`True` if and only if at least one operand is `True`), and `not` (reverses the truth value of the operand).

Selbsttest: Schaut euch das Video 013 an und denkt anschließend über das *Nichts* nach. Ist `False` dasselbe wie `None`? Ist es dasselbe wie `0`?

2.2.3 Integers

The data type `int` represents integer numbers with a sign, i.e., the numbers are not rational and have a sign (can be positive or negative). Examples are 0, 1, −1, 2, 122 42, −1234. Some operations on integer numbers are:

- `(+)`, `(-)`, `(*)` are the classical numerical operations.
- `(/)` divides two integers and returns a floating-point number.
- `(//)` divides two integers and returns an integer.
- `(%)` is the modulo operation. When dividing an integer by another integer, the modulo operation returns the remainder.
- `abs(·)` returns the absolute value of a number.

Many languages also provide **`unsigned integers`**, where we do not have negative integers.

Problems with integers. Integer numbers in Python can be arbitrarily large. At first, this might look like a big advantage. However, when looking deeper into the implementation of Python itself, one can observe that this implementation is ineffective/slow.

Other languages like Scala, C++ or Java do not have arbitrarily large integer numbers. Their integer types have finite precision (32 or 64 bits) but operations can be processed faster. They usually can represent integer numbers in the ranges $[-2^{31}, 2^{31} - 1]$ or $[-2^{63}, 2^{63} - 1]$. The main disadvantage is that whenever the result of an arithmetic expression is not within this range, a so called *overflow error* occurred. We can show this behaviour with Scala:

```
>>> val a : Int = 2147483647     // = 2^31 - 1 (within the range)
>>> val b : Int = a + 1          // = 2^31 (not within the range)
>>> println(b)                   // OVERFLOW ERROR
-2147483648
```

Later in this chapter, we look at the exact representation of integer numbers in languages like Scala or C++.

Selbsttest: Schaut euch das Video 014 an. Worauf muss man beim Programmieren mit ganzen Zahlen im Allgemeinen achten? Gibt es das Problem auch mit Python? Was passiert, wenn man `-13 % 4` rechnet?

2.2.4 Floating-point numbers

Floating-point numbers (`float`) represent *a subset of* the rational numbers, represented in a special binary format (called IEEE754). Some operations on floating-point numbers are:

- `(+)`, `(-)`, `(*)`, `(/)` are the classical numerical operations.
- `int(·)` converts a floating-point number into an integer, by removing the digits after the decimal point.

Many languages distinguish between the types `Float` (32 bits) and `Double` (64 bits).

Problems with floating-point numbers. We have a fixed number of bits (32 or 64) to represent rational numbers. Therefore, we can represent 2^{32} or 2^{64} different numbers. But there are infinitely many different rational numbers. The conclusion is that there are infinitely many rational numbers that cannot be represented by floats or doubles. Moreover, the numbers are represented as binary fractions. In particular, 0.1 is a periodic binary fraction and cannot represented precisely in floating-point arithmetic. That means, when working with floating-point numbers we must become aware that there will always be rounding errors and certain mathematical laws, like the associative law, do not hold any more.

```
>>> 0.1*3
0.30000000000000004 # Suddenly some (very small) error appears
>>> 1+0.0000000000000001
1.0 # Adding this very small number is "ignored", thus it does not help ...
>>> (1+0.0000000000000001)+0.0000000000000001
1.0 # ... trying to add it more often
>>> 1+(0.0000000000000001+0.0000000000000001)
1.0000000000000002 # Adding the two small numbers, gives the desired result
```

Selbsttest: Schaut euch das Video 015 an. Recherchiere anschließend den Begriff „Unterlauf“. Worauf muss man beim Programmieren mit Floats achten?

2.3 Composite data types

Composite data types allow us to collect several simple data items together and to treat them as a unit. This allows us to group data in our program, and to manipulate arbitrarily large data sets. There are several composite data types in Python.

2.3.1 Characters and Strings

The data type `str` represents sequences/*strings* of *characters*, where the characters are taken from a standardized table of possible characters (e.g., standard ASCII, extended ASCII, UNICODE, ...). Python uses UNICODE-strings. Strings are represented using single or double quotes (`"Hallo"`, `'Welt'`, `"Hallo 'Welt'"` , `'"Hallo" Welt'`). Some operations on strings are:

- `(+)` *concatenates* two strings.
- `len(·)` returns the length of a string, i.e., it returns the number of characters.
- `ord(·)` returns the ordinal number of a character. The function produces an error, if its input is not of length 1.
- `chr(·)` is the inverse function of `ord(·)`. It takes an integer and returns the character according to the UNICODE-assignment.

```
>>> s = "Hallo " + "Welt"
>>> s
"Hallo Welt"
>>> len(s)
10
>>> ord("a")
97
>>> chr(98)
"b"
```

In Python a character is the same as a string of length 1. Many languages differ between strings of length 1 and characters. For example in Scala there are the two data types `String` and `Char`. Moreover, in Python strings are *immutable*, i.e., we cannot change them.

Selbsttest: Schaut euch das Video 016 an. Schreibe ein kleines Programm, welches ein Symbol als Eingabe erwartet und dasjenige Symbol auf dem Bildschirm ausgibt, welches in der UNICODE-Tabelle 3 Stellen weiter vorne ist. (Aus `'d'` wird zum Beispiel `'a'`.)

2.3.2 Lists

A list in Python consists of a sequence of data objects. In Python, the objects can be of distinct types. The order of the objects matters, and the same data object can appear repeatedly in a list. For example,

```
xs = [1, True, "Hello", 5.5, 1]
```

defines a list `xs` that consists of 5 elements: the integer number 1, the truth value `True`, the String `"Hello"`, the floating-point number 5.5 and the integer number 1, in this order.

Index notation. We can access individual data items of a list using index notation, starting with index 0. E.g., `xs[1]` is the truth value `True`, `xs[4]` is the integer 1. Moreover, in Python it is possible to use negative index values: they are counted from the back of the list, starting with −1. E.g., `xs[-2]` is the floating-point number 5.5. We will get an error if we try to access an index that does not exist (e.g., `xs[10]` or `xs[-7]`). We can determine the number of elements in a list with `len(xs)`, the possible index values go from 0 to `len(xs)-1` and from −1 to `-len(xs)`.

Lists are *mutable*, that means we can change entries of a list by using the index notation.

```
>>> xs[1]
True
>>> xs[1] = 7
>>> xs
[1, 7, "Hello", 5.5, 1]
```

It is important to note that the assignment statement `xs[1] = 7` changes the existing list `xs`. This means that this change is also visible for other variables that are associated with the same list. For example,

```
>>> ys = xs
>>> xs[1] = 7
>>> ys
[1, 7, "Hello", 5.5, 1]     # ys has changed since xs was changed
```

A list can contain other lists, e.g., we can have the following list

```
xss = [[7,2,3], [1,2], [4,6], [3,4,5,6]]
```

The list `xss` consists of 4 lists, the first has 3 elements, the second has 2 elements, the third has 1 element and the fourth has four elements. We can use a double index notation to access the elements in the sublists. For example, `xss[1][0]` gives 1, the first element of the second list, while `xss[3][2]` gives 5, the third element in the fourth list. With `xss[2]`, we get the whole third list in `xss`, i.e., the list `[4,6]`. We call `xss` a *two-dimensional list*, and of course, there are also three-dimensional and higher-dimensional lists.

Almost every programming language supports a concept of two- or higher-dimensional lists, but it may be more restrictive than in Python.

Attention: Since Python is dynamically typed, it is possible, that a list can have objects of different types. This is not common in other languages. Many languages, like Scala, Java, C++, allow only lists in which the elements are of the same type, i.e., the list `xs` from above would not be possible.

Selbsttest: Schaut euch das Video 017 an. Zeichne das Speicherbild für die Liste `xss`.

Operations. A specialty of Python are *slices*, a special syntax to extract sublists from a given list. The notation is `xs[i:j:k]`. This generates a sublist of `xs` that starts at index `i` (included) and increases the index by increments of `k`, until the first value larger of equal to `j` is reached (excluded). If we omit `i`, it corresponds to 0 the first index in the list. If we omit `j`, it corresponds to the largest index in the list. If we omit `k`, it is taken to be 1. We can also use a negative value for `k`, in which case we go through the list in reverse order.

Further operations on lists include `(+)` (concatenation of two lists into a longer list) and `(*)` (repeating a list a given number of times).

```
>>> 3 * [1,2,3]
[1,2,3,1,2,3,1,2,3]
>>> [4,5,6] + [1,2,3]
[4,5,6,1,2,3]
>>> zs = [3,5,1,4,5,7,8]
>>> zs[1:6:2]
[5,4,7]
```

Lists support many more operations that modify them. For example, the operation `append` adds an element to an existing list:

```
>>> xs = [1,2,3]
>>> ys = xs
>>> xs.append(10)
>>> ys
[1,2,3,10]
```

After this, both `xs` and `ys` are associated with the same list, namely `[1,2,3,10]`. This should be compared to the concatenation operator that does not modify an existing list but creates a new list (and that is also available for tuples). For example, consider the following code:

```
>>> xs = [1,2,3]
>>> ys = xs
>>> xs = xs + [10]
>>> ys
[1,2,3]
```

After this, `ys` still refers to the old list `[1,2,3]`, while `xs` refers to a new list that was created by the assignment operator, namely `[1,2,3,10]`.

Finally, we want to mention that lists can be used in `for`-loops, e.g., we can write `for i in xs:` to iterate over every element that is contained in the list `xs`, in the order in which they appear in `xs`. The program

```
for x in [1, True, "Hello", 5.5, 1]:
    print(x)
```

produces the output

```
1
True
Hello
5.5
1
```

Selbsttest: Schaut euch das Video 018 an. Schreibt ein kleines Programm, welches mit einer Liste `xs = [1,2,3,4,5,6,7,8]` startet, anschließend auf jedes Element der Liste eine `2` addiert und aus der resultierenden Liste die Summe aller Elemente berechnet.

2.3.3 Tuples

Everything we have said so far about lists in Python also holds for tuples in Python, except that tuples are written with round parentheses instead of square brackets. That is,

```
tup = (1, True, "Hello", 5.5, 1)
```

defines a tuple with 5 elements. The index operator for tuples also uses square brackets, i.e., the second element of tuple `tup` is accessed via `tup[1]`.

The big difference between lists and tuples in Python is that lists are *mutable*, whereas tuples are *immutable*. That is, we can make changes to the elements of an existing list, whereas we cannot make changes to the elements of an existing tuple. If we tried to write `tup[1] = 100`, we would get an error.

Selbsttest: Schaut euch das Video 019 an und arbeitet die wichtigsten Unterschiede zwischen Tupeln und Listen heraus. Ist ein String in Python eher eine Liste oder eher ein Tupel?

2.3.4 Dictionaries

Dictionaries are a particularly convenient feature of Python. Dictionaries store *key-value* pairs, where a key can appear at most once and can be used to look up a value. A

key-value pair is called in *item*. For example, we can write

```
dict = {1: "one", 2: "two", 3: "three"}
```

to create a dictionary with three items. The first item has key 1 and value `"one"`, the second item has key 2 and value `"two"`, and the third item has key 3 and value `"three"`.

We can retrieve the value for a given key using the index notation. For example, `dict[2]` retrieves the value for key 2, that is, the string `"two"`.

If we try to access the value for a key that does not exist, we get an error.

Dictionaries are mutable, there are several ways to modify the items that are stored in an existing dictionary. For example, the assignment

```
dict[key] = expression
```

has two possible effects: if `key` already exists in `dict`, the value is changed to the value of `expression`, e.g., we could write `dict[2] = "dos"` to change the value for key 2. Afterwards, `dict[2]` yields `"dos"`. If `key` does not exist in `dict`, we create a new item with key `key`. For example, the assignment `dict[4] = "four"` would create a new item with key 4 and value `four`. To remove an item with key `key`, we write `del dict[key]`.

Any Python object can be used as a value in a dictionary, but not every Python object can be used as a key, e.g., lists and dictionaries cannot be used as keys, but all simple Python types (integers, floats, strings) are possible.

We can use dictionaries in `for`-loops. If we use the dictionary itself as a range, we iterate over the keys of the dictionary (the same happens if we use `dict.keys()`). To iterate over all values, we must use `dict.values()`. To iterate over all items (tuples that represent the key-value pairs in the dictionary), we use `dict.items()`.

```
>>> dict = {1: "one", 2: "two", 3: "three"}
>>> dict.keys()
[1,2,3]
>>> dict.values()
["one","two","three"]
>>> dict.items()
[(1,"one"),(2,"two"),(3,"three")]
```

Selbsttest: Schaut euch das Video 020 an. Was sagt ihr zu folgender Hypothese: „Eine Liste ist nur eine besondere Form des Dictionaries“?

2.4 Number representation

2.4.1 Natural numbers

Typically, we nowadays represent numbers in the *decimal system*. A number in the decimal system can be represented as a sum. For example,

$$73281_{10} = 70000 + 3000 + 200 + 80 + 1.$$

Here, the digit in position i (counted from the right) has positional value 10^{i-1}. That is,

$$73281_{10} = 7 \cdot 10^4 + 3 \cdot 10^3 + 2 \cdot 10^2 + 8 \cdot 10^2 + 1 \cdot 10^0.$$

Instead of 10 we can use any other natural number (larger than 1) as a basis. In computer science, we typically use bases 2, 8, or 16. Thus, a binary number $x = b_{n-1} \dots b_0$ has the value

$$\sum_{i=0}^{n-1} b_i \cdot 2^i.$$

For example,

$$\begin{aligned} 10011_2 &= 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\ &= 16 + 2 + 1 \\ &= 19_{10}. \end{aligned}$$

In a computer, the data values are represented in memory cells with a certain fixed number of binary digits (nowadays, usually 32 or 64). The digits of a binary number are called *bits* (an abbreviation for *bi*nary digi*ts*).

Without sign, the maximum number that can be represented by a 64bit number is $2^{64} - 1$ (all bits are 1).

If, e.g., an addition or multiplication results in a number that has more than the available number of bits, the upper bits are dropped – we got an overflow error.

Selbsttest: Schaut euch das Video 021 an. Bestimme die Binärdarstellung der Zahl 711_{10} und bestimme die Dezimaldarstellung der Zahl 101101101_2.

2.4.2 Integer numbers

In order to represent negative numbers, we use the *two's complement representation*. For this, we assume that we have a fixed number b of bits available ($b = 32$ or $b = 64$ bits).

The integer numbers in $[0, 2^{b-1} - 1]$ are represented binary as described above. The integer number -2^{b-1} is represented by the binary string `10...0`. Finally, we can get the representation of the integer numbers in $[-2^{b-1} - 1, -1]$ as follows:

(i) Start from the representation of the corresponding positive number.

(ii) Invert all the bits.

(iii) Increment the result by 1.

Finally, we have a representation for all integer numbers in $[-2^{b-1}, 2^{b-1} - 1]$. This representation has the following advantages and disadvantages:

+ Each number in the interval has a unique representation.

+ Subtracting a number is the same as adding the inverse number.

- The interval is not symmetric. (There is a negative number more.)

The two's complement representation is the reason that in many programming languages (not Python), we may have an overflow to negative numbers when the numbers we use in our program get too big.

Selbsttest: Schaut euch das Video 021 an. Bestimme das Zweierkomplement mit 6 Bits der Zahlen 26_{10} und -14_{10}. Addiere diese beiden Zahlen (im Zweierkomplement) anschließend und wandle das Ergebnis wieder in eine Dezimalzahl um (Kommt 12 raus?).