1 Python syntax

Code is separated by **whitespace** (spaces) and special characters (e.g. `=`, `:`, `(`, `)`). **Blocks of code** are **indented** by 4 spaces:

```
no indentation
    first indent level
        indent level 2
        still level 2
    back to level one
```

2 Conditionals

Conditionals allow us to **change the program behaviour** depending on variable / input values.

2.1 IF statement

If the condition is true, run the indented code. If the condition is false, skip the indented code:

```
if (city == "Brno"): # Mind the ":"!
    print "Welcome to the seminar!"

if (cityPopulation > 1000000):
    print "What a big city!"
```

2.2 ELSE, (ELIF)

If the first condition is false and **else** statement is present (there doesn't have to be an **else**), the following indented block of code is executed:

```
if (city == "Brno"):
    print "Welcome to the seminar!"
else:
    print "Maybe next time?"

if temperature < 0: # the parethesis don't always have to be there!
    print "Looks like it's quite cold."
    print "Maybe you can go ice skating?"
else:
    print "Water is not frozen yet!"
```

Elif is basically a short way to type `"else: if (condition):"` → `"elif (condition):"`

```
if (temperature > 50):
    print "Stay home if you can."
elif (temperature < -25):
    print "Stay home if you can."
elif (temperature < 10):
    print "Take a jacket."
else:
    print "Enjoy the sunny day (if it's not rainy)!"
```

2.3 Logical operators

We can **extend conditions** and make more complex checks:

```
if (temperature > 50 or temperature < -25): # save two lines of code
    ↪ compared to elif()
    print "Stay home if you can."
elif (temperature < 10):
    print "Take a jacket."
else:
    print "Enjoy the sunny day (if it's not rainy)!"
```

Not reverses the boolean value.

```
haveWeapon = True
policeman = False
polite = True

if (haveWeapon and not polite):
    print "The bank is waiting."
if (policeman and not haveWeapon):
    print "Get a weapon, quick!"
if (polite and policeman):
    print "Watch out near the bank."
```

2.4 Nesting conditions

```
feelingSick = True
money = 150
car = False

if (feelingSick):
```

```
    if (car):
        print "Drive to the hospital."
    elif (money > 100000):
        print "Take a taxi, whatever."
    else:
        print "Take a bus."
else:
    if (car and money > 100):
        print "Plan a trip somewhere."
    else:
        print "At least you are not sick."
```

3 Lists

Data structure useful for storing sequence of values. Each element of the sequence has a position = **index**, the same as with strings! (The first element has index 0 again). Lists can be operated very similarly to strings!

```
## list containing only strings
cities = ["Praha", "Brno", "Ostrava", "Pardubice", "Jihlava"]

## list containing only numbers
coords = [49.1876, 16.3273, 420.3]

## list with mixed value types
row = [0, True, "Brno", 377440, 2, 49.2, 16.6, ["MUNI", "VUT", "MENDELU",
    ↪  "VFU", "JAMU", "UNOB"]]
```

3.1 Accessing list values

Again, the same as with strings:

```
print cities[1] # "Brno" → returns string
print cities[2:4] # ["Ostrava", "Pardubice"] → returns list
```

3.2 Updating list values

We can update list values with the **[] notation** (strings cannot do that!):

```
cities[3] = "Vidnava"
print cities # ['Praha', 'Brno', 'Ostrava', 'Vidnava', 'Jihlava']
cities[3:] = ["Zlín", "Znojmo"]
```

```
print cities # ['Praha', 'Brno', 'Ostrava', 'Zlín', 'Znojmo']
```

list.insert(index, item) – put *item* at selected *index* (and move the rest of the items) → *insert* doesn't remove anything.

```
cities.insert(0, "České Budějovice")
print cities # ['České Budějovice', 'Praha', 'Brno', 'Ostrava', 'Zlín',
    ↪  'Znojmo']
```

list.append(item) – add an *item* to the end of the *list*

```
cities.append("Vrchlabí")
print cities # ['České Budějovice', 'Praha', 'Brno', 'Ostrava', 'Zlín',
    ↪  'Znojmo', 'Vrchlabí']
```

del – remove selected items:

```
del cities[0] # remove first item → ['Praha', 'Brno', 'Ostrava', 'Zlín',
    ↪  'Znojmo', 'Vrchlabí']
del cities[3:] # remove fourth up to the last value
print cities # ['Praha', 'Brno', 'Ostrava']
del cities[0:] # delete all values → []
del cities # delete the whole variable → NameError: name 'cities' is not
    ↪  defined
```

list.remove(item) – remove the first occurrence of *item*:

```
wallet = [100, 100, 200, 500, 5000]
wallet.remove(5000)
print wallet # [100, 100, 200, 500]
wallet.remove(100) # only the first occurrence is removed
print wallet # [100, 200, 500]
```

3.3 Operations with lists

len(list) – return *list* length (just like string).
concatenation with + – merging lists:

```
cities = ["Brno", "Praha"]
castles = ["Veveří", "Trosky", "Špilberk"]
places = cities + castles + ["Česká republika"]
```

value **in** list – returns *True* if a *value* is in a *list*:

```
city = raw_input("What city do you live in? ")
czechCities = ["Praha", "Brno", "Ostrava"] # etc.

if city in czechCities:
    print "Looks like you live in the Czech Republic!"
else:
    print "Welcome to the Czech Republic, hope you like it here!"
```

max(list), **min(list)** – find the maximum / minimum values in a *list*

```
regionsPopulation = [1562, 45847, 15454, 1521, 48572, 876, 6797]
print max(regionsPopulation) # 48572
print max(regionsPopulation) # 876
```

list.count(item) – count how many times *item* appears in a list (string is again the same):

```
students = ["Andrej", "Daniel", "Abdul", "Kristína", "Kristýna", "Martina",
    ↪ "Annamária", "Ondřej", "Simon", "Pavla", "Lukáš", "Hana",
    ↪ "Kristýna", "Jaroslav"]
print students.count("Kristýna") # → 2
```

list.index(item) – returns the index of the first occurrence of *item* in the *list*. This is the equivalent of *string.find(item)* (*find* doesn't work with lists!):

```
students.index("Kristína") # → 3
```

list.reverse() – reverse the whole *list*. This works in place! Also, *reverse* doesn't take any arguments!

```
students.index("Kristína") # → 10
```

3.4 List conversion

string.split(separator) – convert *string* to *list*, where every *list* items are separated by the *separator* in the string.
Useful e.g. for storing first and last names:

```
cities = "Brno, Praha, Ostrava, Znojmo"
cities = cities.split(", ")
print cities # cities variable changed to ['Brno', 'Praha', 'Ostrava',
    ↪ 'Znojmo']
```

Tip

We can use a shorthand variable definition (if we know the name is only two words long, otherwise it won't make sense):

```
name = "George Lucas"
[first, last] = name.split(" ")
```

string.join(list) – converts *list* to string and joins items with the *string*. The argument is the *list* we want to join!

```
route = ['Brno', 'Breclav', 'Bratislava', 'Gyor', 'Budapest']
print "Recommended route is: " + " - ".join(route)
# Recommended route is : Brno - Breclav - Bratislava - Gyor - Budapest
```

3.5 Lists nesting

Useful in GI for rasters or other data storage:

```
# parking area is just a line of parking places
paring_area = ['car1', False, ' False, 'car2', 'car3'] # 1 * 5 parking
    ↪ places
parking_area[2] # accesing car with index 2

# car park has more "lines", is "2D"
car_park = [ ['car1', False, False], ['car2', 'car3', 'car4'], ['car5',
    ↪ False, 'car6'] ] # 3 * 3 parking places
parking _area[0][2] # accesing car with index 2 in "line" 1

raster = [[1, 24, 36], [8, 0, 9], [4, 255, 3]] # 3 * 3 raster

# multi-storey parking building with "3D" parking - floor and "line"
parking_building = [ [ ['car1', False], ['car2', 'car3'] ], [ [False,
    ↪ False], [False, 'car4'] ], [ ['car5', False], ['car6', 'car7'] ] ] #
    ↪ three floors, each with 2 lines with 2 parking slots
parking_building[1][0][1] # getting parking slot on the 2nd floor, 1st line
    ↪ and 2nd position
```

4 FOR cycle

Basic iteration – repeat block of code for every element in a sequence.

We can iterate over a string:

```
alphabet = 'abcdefghijklmnopqrstuvwxyz'
for letter in alphabet:
    # variable letter is declared to be one character of alphabet in every
    ↪ iteration
    # number of iterations is equal to len(alphabet)
    print letter
```

Cycle through a list:

```
invitedPeople = ["Bob", "Alan", "Emily", "Gustav"]
for person in invitedPeople:
    print "Send invitation to", person

## This will print
# 'Send invitation letter to Bob'
# 'Send invitation letter to Alan'
# 'Send invitation letter to Emily'
# 'Send invitation letter to Gustav'
```

Use **range()** to create lists with incrementing numbers:

```
range(5) # → [0, 1, 2, 3, 4]
range(2,8) # → [2, 3, 4, 5, 6, 7]
range(1,18,3) # → [1, 4, 7, 10, 13, 16]
```

Useful for getting not only the value, but its index too:

```
for position in range(len(cities)):
    print "City no.", position + 1, cities[position]
# will print "City no. 1 Praha", "City no. 2 Brno"
```

Conditions inside cycles:

```
polygons = [
[[1,7], [1,3], [2,3], [2,7]], # polygon 1
[[1,1], [1,5], [3,3]], # polygon 2
[[0,0], [0,5], [2,10], [4,7], [3,10], [8,2]] # polygon 3
]

for polygon in polygons:
    if len(polygon) < 3:
        print "This is not a polygon!"
```

```
    elif len(polygon) == 3:
        print "This is a triangle"
    elif len(polygon) == 4:
        print "This has four sides (might be a square?)."
    else:
        print "This is a more complex polygon."
```

Creating a line with random coordinates:

```
import random # Loading some library for generating random numbers

length = 10 # for example
length = int(raw_input("How long should the line be? ")) # or we can ask
    ↪ the user!

line = []
for i in range(length):
    coordinates = [random.random() * 1000, random.random() * 1000]
    line.append(coordinates)

print line
```