1 Functions

It's great that we can now use Python to find out lengths of lines, areas of polygons and generate sample lines, if only we didn't have to repeat the same code whenever we want to do something …

Oh, wait:

```python
from random import random

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

def calculateLength(line):
    length = 0
    for i in range(len(line) - 1):
        length += ((line[i][0] - line[i+1][0]) ** 2 + (line[i][1] -
            ↪  line[i+1][1]) ** 2) ** 0.5
    return length

# We use it like this:
line = randomLine(5)
# or:
print calculateLength(randomLine(10)) # e.g. 606.7983930932138
```

The function definition consists of:

- `def` keyword
- **function's name** – similar to a variable name
- function's expected **arguments** in parentheses – we can name them how we want (similar to `item` in this for loop: `for item in items:`)
- **code** of the function in an indented block using the function's arguments
- `return` statement containing the result of the function

Some simple examples:

```python
fridge = {
    "milk": 5,
    "eggs": 4,
    "cakes": 0
}
```

```
def buyEggs():
    fridge["eggs"] += 3

def buyMilk():
    fridge["milk"] += 1

def makeCake():
    if (fridge["milk"] >= 2 and fridge["eggs"] >= 4):
        fridge["milk"] -= 2
        fridge["eggs"] -= 4
        fridge["cakes"] += 1
        print "We have a new cake in the fridge!"
    else:
        print "You don't have enough ingredients for a cake!"
```

1.1 Arguments

We can pass values to alter the function's behaviour with **arguments**.

```
def buyEggs(amount):
    print "Fridge before:", fridge
    fridge["eggs"] += amount
    print "Added", amount, "eggs into the fridge."
    print "Fridge after:", fridge

# and call it like this:
buyEggs(5)
# Fridge before: {'cakes': 0, 'eggs': 4, 'milk': 5}
# Added 5 eggs into the fridge.
# Fridge after: {'cakes': 0, 'eggs': 9, 'milk': 5}
```

A good way to define functions is to set **default values** to arguments. This value will be applied if the argument is not passed:

```
def buyEggs(amount = 6):
    print "Fridge before:", fridge
    fridge["eggs"] += amount
    print "Added", amount, "eggs into the fridge."
    print "Fridge after:", fridge

buyEggs()
```

```
# Fridge before: {'cakes': 0, 'eggs': 4, 'milk': 5}
# Added 6 eggs into the fridge.
# Fridge after: {'cakes': 0, 'eggs': 10, 'milk': 5}

buyEggs(5)
# Fridge before: {'cakes': 0, 'eggs': 10, 'milk': 5}
# Added 5 eggs into the fridge.
# Fridge after: {'cakes': 0, 'eggs': 15, 'milk': 5}
```

We might also have more refrigerators and want to do the whole shopping in one function:

```
fridges = {
    "John": {
        "eggs": 1,
        "milk": 2,
        "cakes": 0
    },
    "Ellen": {
        "eggs": 0,
        "milk": 1,
        "cakes": 3
    }
}

def buyGroceries(fridge, groceries):
    for item in groceries:
        if not item in fridges[fridge]: # in case we buy e.g. oranges
            fridges[fridge][item] = groceries[item]
        else:
            fridges[fridge][item] += groceries[item]

buyGroceries("John", {"eggs": 10, "milk": 2, "oranges": 2})
```

We can also use `*args` and `**kwargs` (**arguments** and **keyword arguments**) if we e.g. don't know **how many arguments** we should expect (or to make it simpler):

```
def buyGroceries(fridge, **kwargs):
    for key in kwargs:
        if not key in fridges[fridge]:
            fridges[fridge][key] = kwargs[key]
        else:
            fridges[fridge][key] += kwargs[key]

buyGroceries("John", eggs=10, milk=2, oranges=2) # keywords are
    ↪  automatically handled as strings
```

1.2 Variable scope – local vs. global

- variables declared **inside functions** are not defined outside – they are called **local variables**
- **global variables** are defined outside functions (or using `global` keyword) and can be read from anywhere

This is often confusing!

```
1  color = "blue"
2
3  def changeColor(newColor):
4      color = newColor
5      print "Color changed to", color
6
7  changeColor("pink")
8  print color # blue
```

See, in the above code, `color` variable on line 1 is a **global** variable, but the `color` variable on line 4 is a local variable and does not change the value of the global one.
But it gets worse …

```
color = "blue"
colors = {
    "background": "gray",
    "foreground": "teal"
}

def changeColor(newColor):
    color = newColor
    colors["background"] = newColor
    print "Colors changed to", color

changeColor("pink")
print color # blue
print colors["background"] # pink - what?
```

One “doesn’t” work and the other does? Let’s not bother with that now and use `global` keyword when you need to change global variables inside functions:

```
color = "blue"
colors = {
    "background": "gray",
    "foreground": "teal"
}

def changeColor(newColor):
    global color
    global colors
    color = newColor
    colors["background"] = newColor
    print "Colors changed to", color

changeColor("pink")
print color # pink
print colors["background"] # pink
```

1.3 Return

Because you can't access **local** variables outside their scope (the function they were defined in), we use `return` statements:

```
def mean(numbers):
    avg = float(sum(numbers))/len(numbers)
    return avg

print mean([1,2]) # 1.5
print avg # NameError: name 'avg' is not defined → avg is a local variable
```

Note

We don't really need local variables for simple functions:

```
def mean(numbers):
    return float(sum(numbers))/len(numbers)
```

2 Import

Import is easy, we use either:

- `import module`:

```
import random
print random.random()
```

- `from module import function1, function2, ...`:

```
from random import random, randint
print random(), randint(1,10)
```

or like this:

```
from random import *
# be very careful using this
```

We can also use this to give structure to our project. Let's save this code example as `lineTools.py`:

```
from random import random
def randomLine(length):
    line = []
    for i in range(length):
        line.append ([random() * 1000, random() * 1000])
    return line
def calculateLength(line):
    length = 0
    for i in range(len(line) - 1):
        length += ((line[i][0] - line[i+1][0]) ** 2 + (line[i][1] -
            ↪ line[i+1][1]) ** 2) ** 0.5
    return length
```

Now, you can import the content of this script to another script **saved in the same directory**: `script.py`

```
import lineTools
print lineTools.randomLine()
```

Important: In PyZo, you have to run the code **as a script**! That is, `Ctrl+Shift+E` instead of `Ctrl+E`.

3 JSON, GeoJSON

3.1 JSON

JSON is a file format used frequently to send data over the internet (`https://en.wikipedia.org/wiki/JSON#Data_types.2C_syntax_and_example`). It uses **similar** data types as Python (numbers, strings, booleans, lists, dictionaries).

JSON example:

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    },
    {
      "type": "mobile",
      "number": "123 456-7890"
    }
  ],
  "children": [],
  "spouse": null
}
```

In Python, the `json` module is useful for processing JSON data:

```
import json
data = {
    "firstName": "John",
    "lastName": "Smith",
    "isAlive": True,
    "age": 25
}
jsonData = json.dumps(data) # convert to a string formatted as JSON
print jsonData
print json.loads(jsonData) # convert to data structure from JSON string
```

3.2 GeoJSON

GeoJSON format specifications: `https://tools.ietf.org/html/rfc7946`

Subset of JSON format used to handle spatial data, example:

```json
{ "type": "FeatureCollection",
  "features": [
    { "type": "Feature",
      "geometry": {
        "type": "Point",
        "coordinates": [102.0, 0.5]
      },
      "properties": {
        "prop0": "value0"
      }
    },
    { "type": "Feature",
      "geometry": {
        "type": "LineString",
        "coordinates": [
          [102.0, 0.0], [103.0, 1.0], [104.0, 0.0], [105.0, 1.0]
        ]
      },
      "properties": {
        "prop0": "value0",
        "prop1": 0.0
      }
    },
    { "type": "Feature",
      "geometry": {
        "type": "Polygon",
        "coordinates": [
          [ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0],
          [100.0, 1.0], [100.0, 0.0] ]
        ]
      },
      "properties": {
        "prop0": "value0",
        "prop1": {"this": "that"}
      }
    }
  ]
}
```

The root element is always a single GeoJSON object. It's type is one of the following:

- Point, MultiPoint
- LineString, MultiLineString
- Polygon, MultiPolygon
- GeometryCollection
- Feature
- FeatureCollection

Read through the specification and the examples at `https://en.wikipedia.org/wiki/GeoJSON`.

You can save data as GeoJSON from QGIS (right click on a layer → Save As) or using other tools (`https://mygeodata.cloud/converter/shp-to-geojson`).