Python for ArcGIS. Lab 1.

Python is relatively new language of programming, which first implementation arrived around early nineties of the last century. It is best described as a high level and general purpose language, that places emphasis on ease of code understanding and possibility of expression of an idea while being size conservative. It allows to write structural programs, as well as object oriented ones, along with possibility of dividing code into smaller parts, which can be imported as modules. Python is considered as easy to learn, as its syntax is a natural English derivative, and it has significantly less special cases than other languages. Additionally scripts written in this language can be run without modification on multiple operating systems.

1. Python basics

To get acquainted with language basics we will going to use Python console available thorough IDLE integrated development environment (IDE). It is located in Start Menu inside ArcGIS → Ptython 2.7 submenu. It allows for sending direct expressions into interpreter. In case where there will be many expressions at once to evaluate we are going to write scripts. For that purpose integrated script editor inside IDLE will be used, but any text editor can be utilized for this purpose.There is special character "#" which means any text behind it is ignored by Python interpreter. It shoul be also noted that almost any data used in Python is an object, even simple string type. What this entails is that they can have various methods defined for them.

Python reference documentation can be found here: https://docs.python.org/2/reference/

1.1 Data types, variables, operators and data agregates

Most programming languages require variables to be declared before usage. Python deemed that unnecessary therefore declaration occurs dynamically in the moment of data assignment. Basic data types are integers (int), floating point numbers (float), and character strings (str). Data, or object, assignment happens thorough usage of "=" operator. Assignment examples can be run in Python console entering expressions below:

```
a=1         # assingment of object int „1” to variable a
b="test"    # assingment of object str to variable b
c=1.0       # assingment of object float to variable c
```

Next the type of defined variables can be verified using **type()** function:

```
type(a)
```

Displayed information should equal to types of int, str and float accordingly. It is worth noticing that floating number can be forced thorough comma operator, even if value which is assigned technically is an integer. Data assigned to variables are objects, and variables are references to those objects. Simple data types are immutable, therefore their content can not be changes. Hence during repeated assignment to the same variable its current data is discarded and new object is created. It has repercussions in case of strings for example, where its fragments cant be modified. To achieve such a goal new object shall be created which will contain selected substrings. This operation is called **slicing** and square brackets are used to that effect as subscripts. Also it is worth mentioning that we count them from **zero** .

```
'Kate has a cat'[3:8] # slice out characters 4th to 8th from object
b[:2]                # slice from beginning to positon 2 (from zero)
('Our '+b)[-6:]      # slice starting from 6 characters counting backwards
```

In Python there os also special object called **None** which assignment of means that variable does not have any value, therefore does not reference any object. String data type can be marked thorough single quote or double quote. Stings, as an object, contain dozens of methods, which can be examined at following address: https://docs.python.org/release/2.5.2/lib/string-methods.html

```
b.capitalize()    # Capitalizes string
'Kate has a cat'.find('cat') # returns 'cat' substring position
```

Apart from simple types such as numbers and strings there are aggregate types available such as **list** (ordered), **set** (unordred) or **dictionary**, and their non mutable versions, namely **tuple** (ordered) and **frozenset** (unorderd). It should be noted that lists can contain different simple data types.:

```
lista=['wpis 0', 'wpis 1', 2, 'wpis 3', 2.0]
zbior=set(('wpis 0', 'wpis 1', 2, 'wpis 3', 2.0))
tupla=('wpis 0', 'wpis 1', 2, 'wpis 3', 2.0)
zbiorz=frozenset(('wpis 0', 'wpis 1', 2, 'wpis 3', 2.0))
slownik={0: 'wpis 0', 1:'wpis 1', 'drugi':2, 'trzeci':'wpis 3', 4:2.0}
```

To display defined assignments we can use **print** operator and **type()** function:

```
print lista
print zbior
print tupla[3]
print slownik['trzeci']
type(zbiorz)
```

After displaying defined variables it is easy to notice difference between ordered and unordered data aggregates. On the other hand difference between mutable and non mutable versions are far less obvious, but they become clear once we take into account, for example, that dictionary structure requires key values, which have to be unique as well as immutable.

Other operators can be applied to objects and variables which are referencing them, for example arithmetic:

```
2+5   # addind int objects
a-3   # substraction of and int variable and int object
a*c   # multiplication of int and float types
c/2   # division of float variable and integer object
```

Without assignment operator console interpreter will display value by default, however it isn't exactly equivalent to **print** operator. It should be noted that when artithmetic operations are performed on two distinct data types such as **int** and **float** then **float** type data is returned. Addition and multiplication operations also can be applied to the strings:

```
″abc″+' aaa'      # string addition
b*3               # string multiplication
b[:1]+b[2:]       # slicing out second character from string
```

Also slicing can be performed on ordered lists, similar to strings:

```
tupla[2:3]
lista[3:]
```

Apart form arithmetic operators there are also comparison and logical ones. Comparison operators allow to check if and how objects differ from each other:

```
10 == 10
10 != 10
10 >= 3
23 < 3
'cat' == 'dog'
'cat' != 'dog'
```

Outcome of those expressions will answer **true** or **false**, otherwise known as logical value of 0 (False) or 1 (True). Logical value can be joined according to the **truth table:**

And	Or	Not
`1 and 1 = 1`	`1 or 1 = 1`	`not 1 = 0`
`1 and 0 = 0`	`1 or 0 = 1`	`not 0 = 1`
`0 and 1 = 0`	`0 or 1 = 1`	
`0 and 0 = 0`	`0 or 0 = 0`	

Therefore example expressions containing those elements can look like this:

```
10 == 10 or 10>=3 # true, because one condition is true
3 < 4 and 4 < 10  # true, because both condifions are true
3 < 4 < 10        # true, shorter notation of the above
6 < 4 < 10        # false, one of the conditions is false
```

There are sometimes situations, when we have to display characters which cant be entered directly form keyboard, because they are not printed or conflict with language syntax. It occurs most often when we need to break a new line, or display quotes. To do this we have to resort to one of the available **escape sequences**. Most popular are:

```
\n    ASCII linefeed (LF) - new line
\r    ASCII Carriage Return (CR) - second character of new line (Windows CR/LF)
\'    single quote
\"    double quote
\\    backslash (\)
\a    ASCII bell (BEL)
\b    ASCII backspace (BS)
\t    ASCII Horizontal Tab (TAB)
```

Example string representing usage of escape sequence can look like this:

```
print “ Kate has a cat \n by the name of \”Snowball\” “
```

Different situation in which we need additional control is displaying variable values as a part of text strings. In this case we can use formatting operator, which in connection with conversion types is interpreted by **print** operator:

```
zmienne = ('test', 1323, 21.432)          # defining tuple
print 'Testing var display: string %s, number int %d, float %f' % zmienne
```

From example we can deduce that **print** operator is using consequent values from list, and interpreting them relaying on formatting operator and then displaying it. It can be enhanced further through usage of **dictionary** which has strings as keys:

```
slow={'k0':'test', 'k1':1323, 'k2':21.432}   # define dictionary
print 'Numbers float %(k2)f, int %(k1)d, and string %(k0)s' % slow
```

Also when we need to just display % character without formatting we can use „%%”.

```
print('Testing %%d %d %%d') % 3
```

Complete conversion table and additional flag values can be found following this address: https://docs.python.org/release/2.5.2/lib/typesseq-strings.html.

1.2 Scripts and flow control

The combination of different commands in one program requires you to create a text file containing those commands, and depending on the language that is used appropriate formatting rules must be respected. In the case of Python source code files traditionally have the extension ".py", but properly formatted file with a different extension also executes correctly. Such files are usually called scripts, because its complation to machine code is not required before execution, if the computer on which they are written, has a given language interpreter installed. During the first execution of the script Python interpreter creates a file with the extension ".pyc", which contains a compiled version of the script, and it is used as long as the source code in the script will not change. You can force a recompile by deleting the file ".pyc".

Python syntax does not require parentheses, colons or are the keywords to mark the program block. Used for this purpose is suitably adjusted indentation, by adding a space at the beginning of the line. That means properly written program is at the same time well formatted and aesthetically pleasing. The script file can be created using any editor, for example, a standard notepad. However, to get the syntax highlighting is good to use another program such as Notepad++. The first simple program can be written by combining the following expressions:

```
a=9
b=3
wyn=9+3
print 'Addition  ',a,' i ', b, ' : ', wyn
```

To create such script we are going to use IDLE and its built-in editor, by selecting File → New Window. Then we have to enter our experessions, and upon completion, we have to save file, and then run the script using menus Run → Run module, or by pressing F5 function key.

Before beginning of further program modification, it is worth noting that the variables of the python in principle can have any name, if it does not interfere with the reserved keywords, namely:

```
and as assert break class continue def del elif else except exec finally for
from global if import in is lambda not or pass print raise return try while
with yield
```

In addition to these keywords standard implementation includes of course a lot of functions and methods, but they have parenthesis after the name, so there is no risk of conflict.

In virtually every program there is a place where, according to some values we want to make the program accomplish another operation than the default. In this situation, we will use the structures ***'if..else'.*** It is single most important construct you can think of, as without it there would not be possible to write programs as we know it:

```
print 'Do you want to continue (t/n)?'
k=raw_input()
if k == 't':
      print 'Going through with execution'
elif k == 'n':
      print 'Ending program'
      exit()
else:
      print 'Wrong key pressed'
```

New used function raw_input() reads the entered value from the keyboard, treating entered values as a string without evaluating its contents.

Admittedly a command from the first script may be entered manually in the console of Python, but often we have to do a lot of similar operations, where manual insertion would be too burdensome. With the help comes construction of the loop, where you can use one of the two: while or for. While construction is carried out as long as the controlling condition is true :

```
print 'Guess number 1 to 9'
while input() != 8:                # is numner correct
      print 'Wrong answer!'        # if not
print 'Bingo!'                     # if yes
```

New function input () reads the entered value from the keyboard and evaluates it according to Python rules, then it is checked whether the value is different from eight. The loop will execute as long as this condition is true. Attention should be placed on the aforementioned indentation, which in this case controls which part of the code is performed in the loop.

Another version of the loop in the language is construction 'for' and unlike 'while' does not use condition, only a defined range of values:

```
print 'Enumerating values 0-99: ',
for i in range(100):
      print i,' ',                 # wyświetl kolejną liczbę
print '\nKoniec enumeracji.'
```

It should be noted at the end of the operator '**print**' there is a comma. This means that we do not want the end of the line displayed. You may also note that the argument of the operator '**print**' appeared as a sign '**\n**'. This is called the escape character and allows you to display characters that are not on the keyboard, for example. end of the line, as is the case here. A new operator 'in' checks whether the value is in the set. New function 'range ()' creates a list of numerical values of the specified range, and it has two possible forms, one we used **'range(**stop**)'**, but it can be called with 3 arguments **'range(**start,stop,step**)'**:

```
range(100, 10, -5)      # generate list of decreasing values
```

Expanding the program further, we can conclude that some portions are too long, and we would like to break them into smaller pieces. We will use then the structure function, which is a separate block that can be called from anywhere and, if necessary, receive parameters:

```
def dodaj(a,b):   # function definition
      c=a+b
      return c    # returns value

print dodaj(2,3)  # beginning of the main program, function call
wyn=dodaj         # function can be referenced by variable as well
print wyn(3,4)    # function call through variable
```

We can also create functions that have optional parameters:

```
def mnoz(a,b=None):     # function definition
      if b==None:
            c=a*a
      else:
            c=a*b
      return c    # returns value

print mnoz(2,3)   # beginning of the main program, function call
print mnoz(4)

wyn=mnoz
print wyn(3)
print wyn(32,12)
```