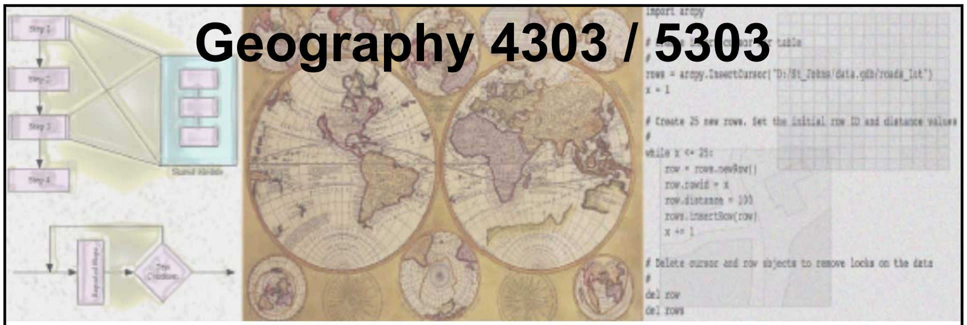


```
import arcpy

# Create insert cursor for table
#
rows = arcpy.InsertCursor("D:/St_Johns/data.gdb/roads_lt")
x = 1

# Create 25 new rows. Set the initial row ID and distance values
#
while x <= 25:
    row = rows.newRow()
    row.rowid = x
    row.distance = 100
    rows.insertRow(row)
    x += 1

# Delete cursor and row objects to remove locks on the data
#
del row
del rows
```

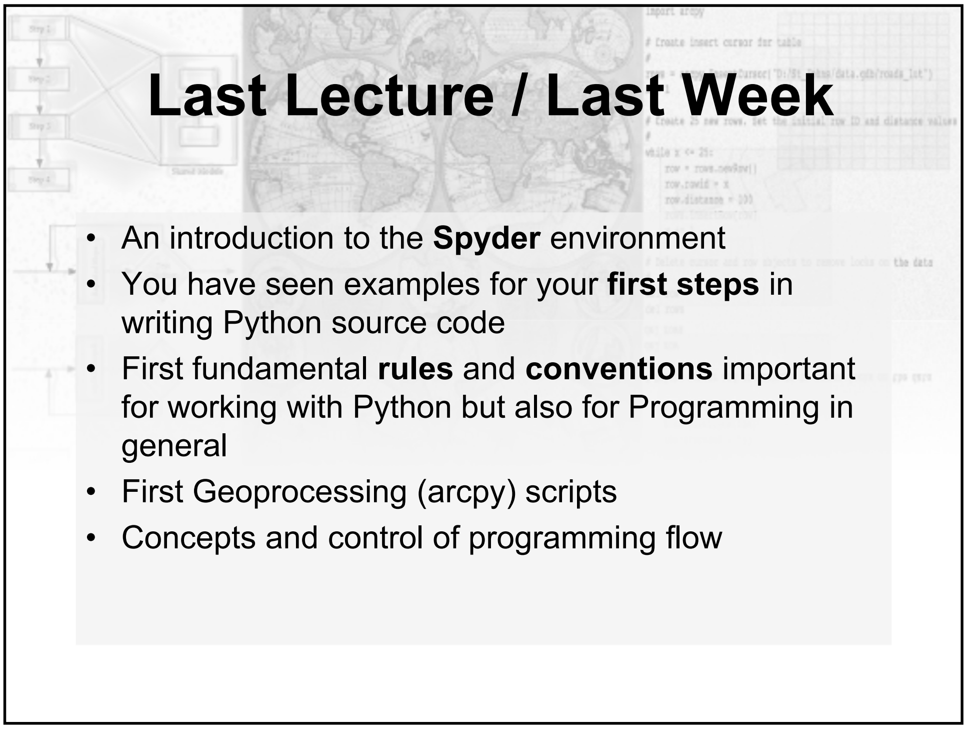
Geography 4303 / 5303

GIS PROGRAMMING FOR SPATIAL ANALYSIS

Class 02: Variables, Data & Data Objects

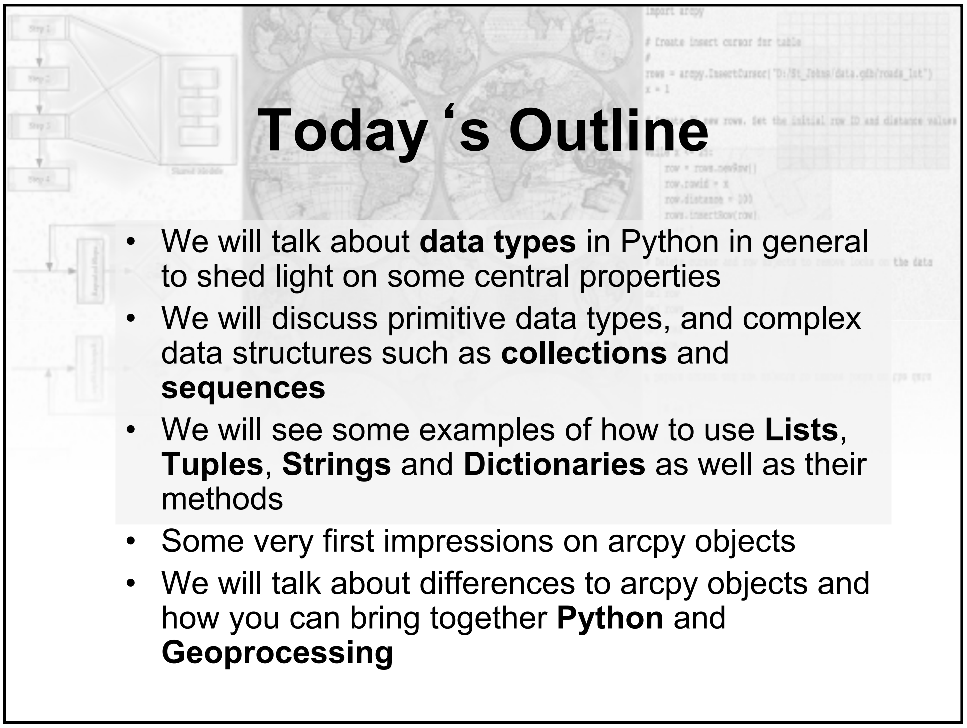
Some Updates

- Sample codes available for download
- Reminder: Project proposals … and procedure
- First lab exercises and demos… any questions?



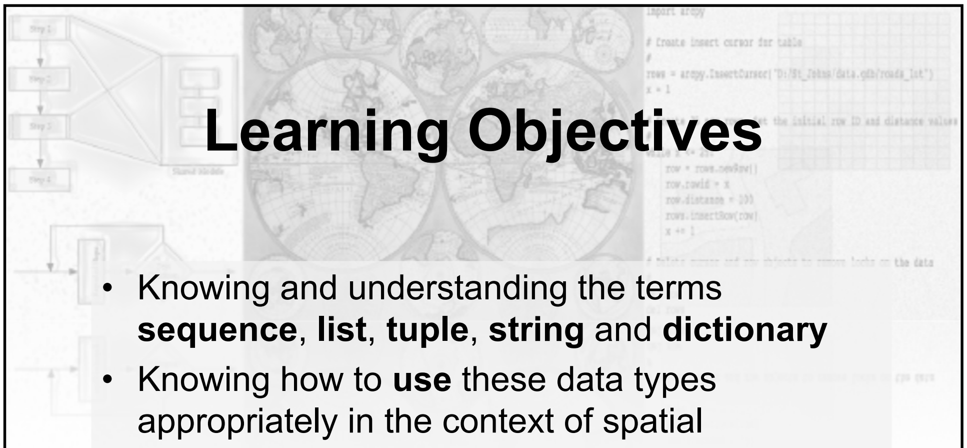
Last Lecture / Last Week

- An introduction to the **Spyder** environment
- You have seen examples for your **first steps** in writing Python source code
- First fundamental **rules** and **conventions** important for working with Python but also for Programming in general
- First Geoprocessing (arcpy) scripts
- Concepts and control of programming flow



Today ‘s Outline

- We will talk about **data types** in Python in general to shed light on some central properties
- We will discuss primitive data types, and complex data structures such as **collections** and **sequences**
- We will see some examples of how to use **Lists**, **Tuples**, **Strings** and **Dictionaries** as well as their methods
- Some very first impressions on arcpy objects
- We will talk about differences to arcpy objects and how you can bring together **Python** and **Geoprocessing**



Learning Objectives

- Knowing and understanding the terms **sequence**, **list**, **tuple**, **string** and **dictionary**
- Knowing how to **use** these data types appropriately in the context of spatial programming and Geoprocessing
- Learning an important **foundation** for programming in general and for Python in particular

The Python library

- Contains **data types** (numbers, lists)
- **Built-in functions** (no import) and exceptions:

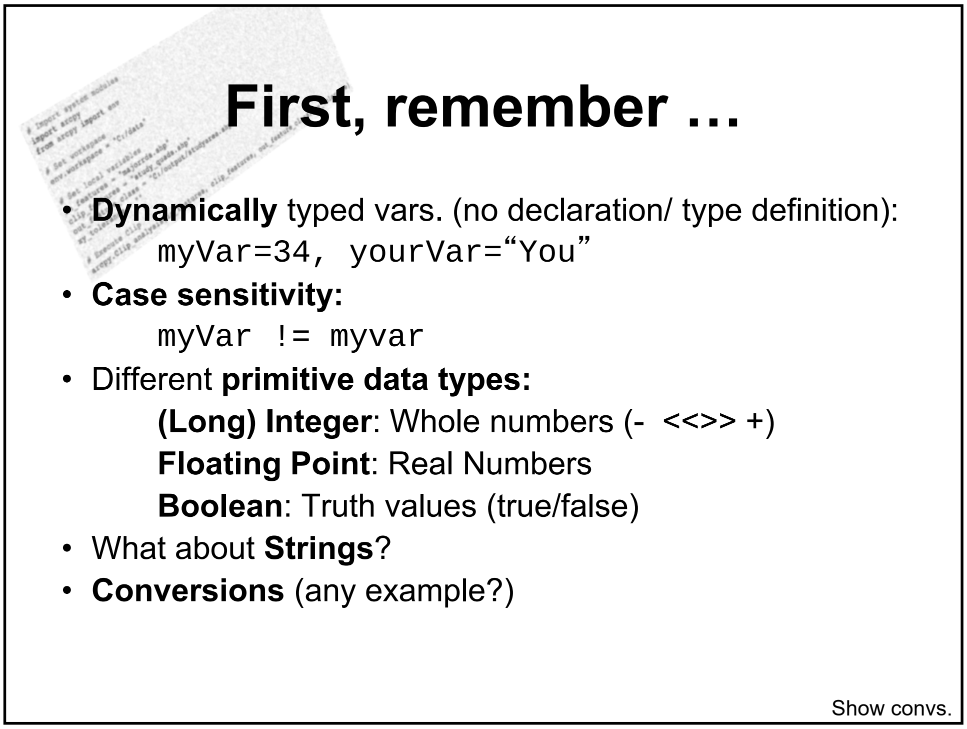
```
myList.append(), type()
```

- Collection of **modules** (to be imported) –

```
import arcpy, math
```

- These can be **YOUR** modules, too:

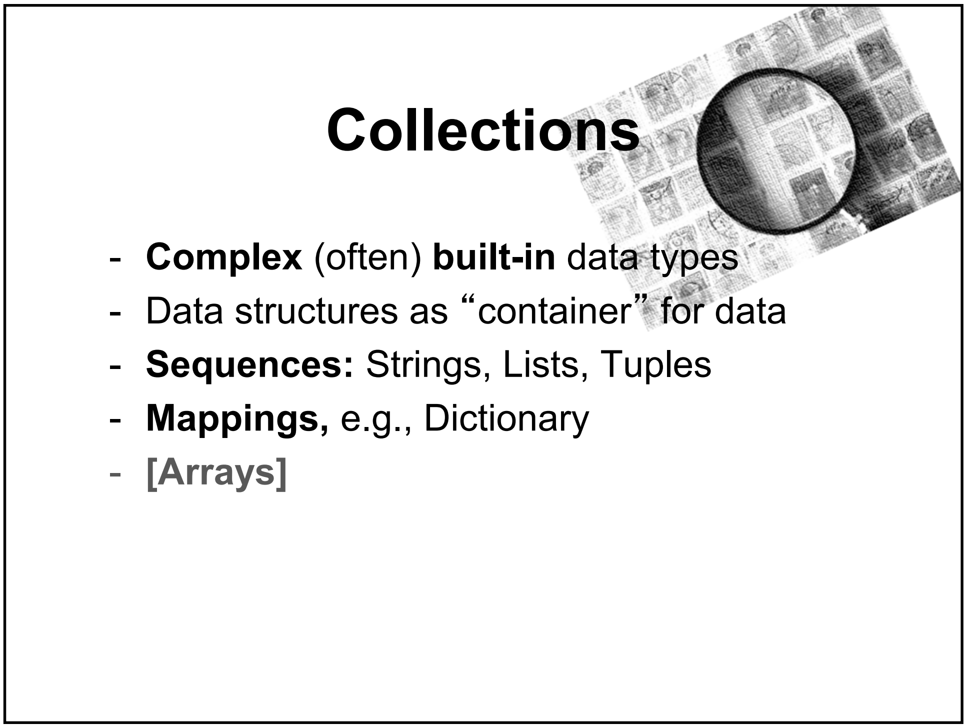
```
import myModule
```



First, remember …

- **Dynamically** typed vars. (no declaration/ type definition):
  `myVar=34, yourVar="You"`
- **Case sensitivity:**
  `myVar != myvar`
- Different **primitive data types:**
  - **(Long) Integer**: Whole numbers (- <<>> +)
  - **Floating Point**: Real Numbers
  - **Boolean**: Truth values (true/false)
- What about **Strings**?
- **Conversions** (any example?)

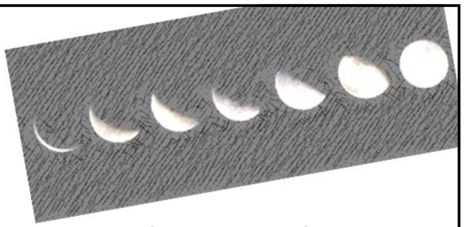
Show convs.



Collections

- **Complex** (often) **built-in** data types
- Data structures as "container" for data
- **Sequences:** Strings, Lists, Tuples
- **Mappings,** e.g., Dictionary
- **[Arrays]**

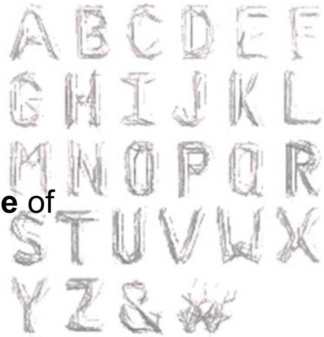
Sequences



- Sequences are data structures that contain **collections** of data
- Allow data handling e.g., **indexing** and **slicing**
- **Strings**, **Lists** and **Tuples** are examples of sequences

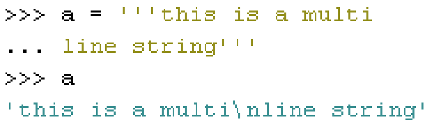
```
myList[2]     #indexing using position
myList[2:6]   #slicing to retrieve part
              #of the sequence
```

Strings 1

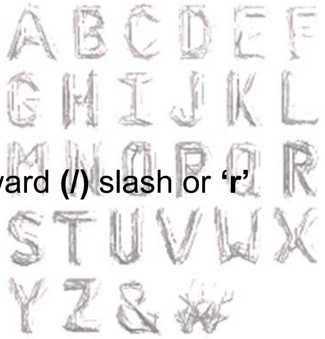


- Data structure that holds a **sequence** of **individual characters**
- `string[character number]`
- **Immutable**
- **Single-line**: Single (‘) or double (“) quote

  `myString=‘Some chars’` `yourString=“More chars”`
- **Multi-line**: Triple (‘ ‘ ‘) or (“””) quote
- Why are Strings important to us?



```
>>> a = '''this is a multi
... line string'''
>>> a
'this is a multi\nline string'
```

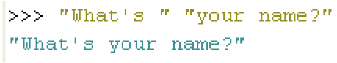


Strings 2

- **Path names**: two back **(\\)**, one forward **(/)** slash or **‘r’**

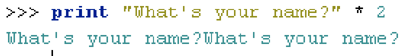
```
path = “c:\\TempDir”
file = “forest.shp”
fullPath = path + “\\” + file
path[3:7] ➔ “Temp”
```

- Automatic **concatenation** of two string literals



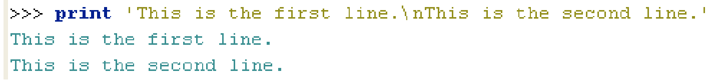
```
>>> "What's " "your name?"
"What's your name?"
```

- **String repetition**

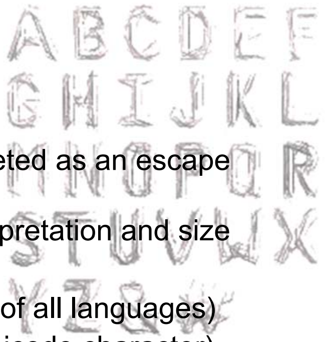


```
>>> print "What's your name?" * 2
What's your name?What's your name?
```

- **Escape sequences**: `\n  \r  \a`



```
>>> print 'This is the first line.\nThis is the second line.'
This is the first line.
This is the second line.
```



Strings 3

- “**raw**” Strings

Changes interpretation: backlash not interpreted as an escape character:

`a=r"e\tl" vs. a="e\tl"` different interpretation and size

- “**unicode**” Strings: 2 bytes (all characters of all languages)

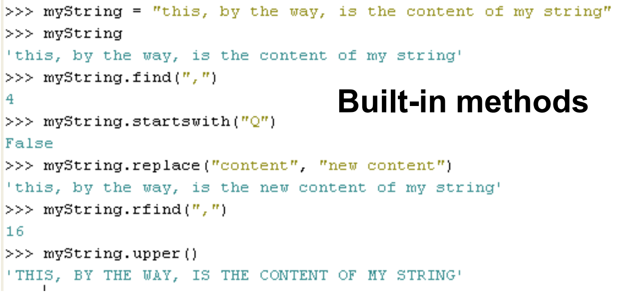
`‘u’` in front creates newer unicode type (Unicode character)

e.g., α - “Miscellaneous Technical” Unicode, turns into a question mark (https://www.ltg.ed.ac.uk/~richard/unicode-sample.html)

- str type stores 8-bit characters – ASCII (1 byte – latin characters and simple symbols)

- str cannot read all Unicode characters! And each character takes less memory

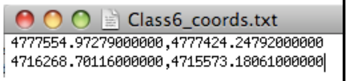
Strings 4



```
>>> myString = "this, by the way, is the content of my string"
>>> myString
'this, by the way, is the content of my string'
>>> myString.find(",")
4
>>> myString.startswith("Q")
False
>>> myString.replace("content", "new content")
'this, by the way, is the new content of my string'
>>> myString.rfind(",")
16
>>> myString.upper()
'THIS, BY THE WAY, IS THE CONTENT OF MY STRING'
```

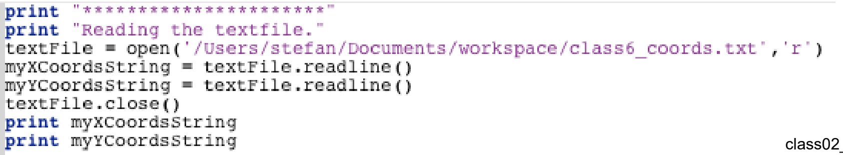
Built-in methods

- Where do we use Strings?
- Pathnames “`c:\\TempDir`”
- Arguments in GP methods: `gp.method(args)`
- Reading, writing file contents



Class6_coords.txt

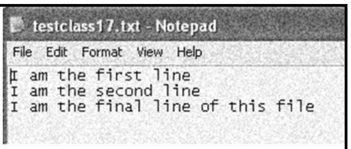
```
4777554.97279000000,4777424.24792000000
4716268.70116000000,4715573.18061000000
```



```
print "**********************"
print "Reading the textfile."
textFile = open('/Users/stefan/Documents/workspace/class6_coords.txt','r')
myXCoordsString = textFile.readline()
myYCoordsString = textFile.readline()
textFile.close()
print myXCoordsString
print myYCoordsString
```

class02_02_textfile.py

Reading Input Files



testclass17.txt - Notepad

```
I am the first line
I am the second line
I am the final line of this file
```

- **`Open`**`( )` the file and flag reading access`(“r”)`: File location as **String** argument:

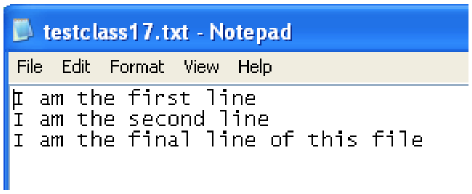
  `myFile = open(‘file.txt’,‘r’)`
- **Read** the first line:

  `myFile.readline()`
- **New line** character is read, too. Need to strip off “\n”:

  `firstLine = firstLine[:-1]`
- **`Close()`** the file when you are done:

  `myFile.close()`

Example Reading Input File



```
I am the first line
I am the second line
I am the final line of this file
```

```
textFile = open('C:/TempDir/testclass17.txt','r')
textLine = textFile.readline()
print textLine, "including new line char"
textLine = textLine[:-1]
print textLine, "without new line char"

textFile.close()
```

```
I am the first line
including new line char
I am the first line without new line char
```

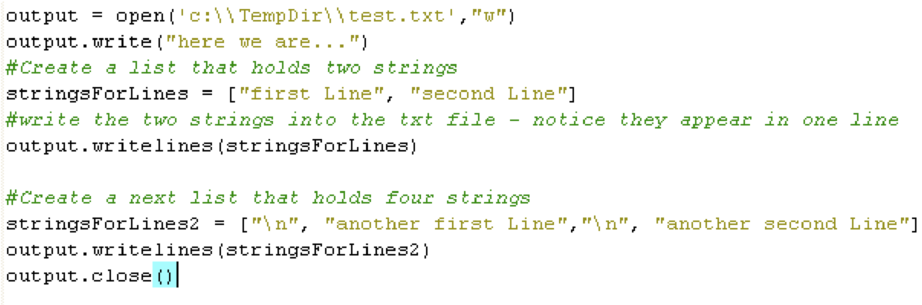
coords through .split()

Writing to Output Files

- **Open( )** the file and flag writing access (“w”): File location as **String** argument : `myFile=open(‘file.txt’,‘w’)`
- **Write** the first line: `myFile.write(‘Something odd’)`
- OR write several lines: `myFile.writelines(myListOfStrings)`
- The **new line** “\n” character is needed if each List element will be inserted in a line
- **Close()** the file when you are done: ▯`myFile.close()`

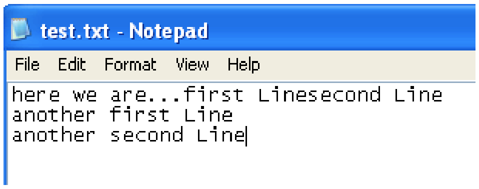
class02_02_textfile.py

Example Writing to Output Files



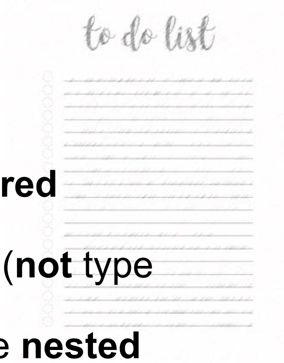
```
output = open('c:\\TempDir\\test.txt',"w")
output.write("here we are...")
#Create a list that holds two strings
stringsForLines = ["first Line", "second Line"]
#write the two strings into the txt file - notice they appear in one line
output.writelines(stringsForLines)

#Create a next list that holds four strings
stringsForLines2 = ["\n", "another first Line","\n", "another second Line"]
output.writelines(stringsForLines2)
output.close()
```

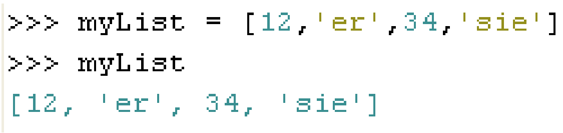


```
here we are...first Linesecond Line
another first Line
another second Line
```

List 1

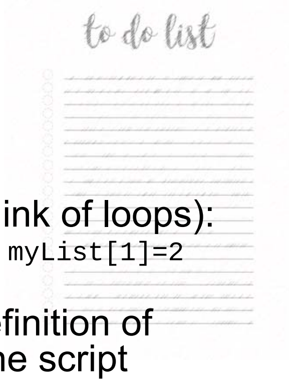


- Data struture that holds an **ordered collection** of items
- Items can be of **different types** (**not** type constrained)
- **Mutable** and several lists can be **nested**
- **Indexed** (starting with index = 0) and referenced as `list[element number]`



```
>>> myList = [12,'er',34,'sie']
>>> myList
[12, 'er', 34, 'sie']
```

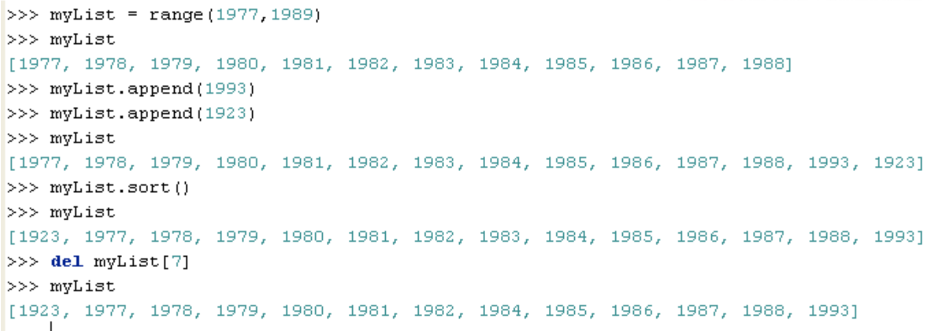
List 2



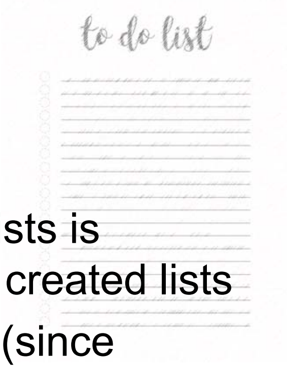
- Created by **direct** assignment (think of loops):
  `myList[1,2,...,x]` `myList[0]=1 myList[1]=2`
- Or: using the **range** function – definition of size can be done while running the script
  ```
  years = range(1976,2008)       years = range(a,b)
  years[3] = 1979
  ```
  `Range()` starts at 1976 and increments up to, but **not** including 2008
- Variables can hold Lists

List 3

- The **object** list also allows the use of built-in methods:

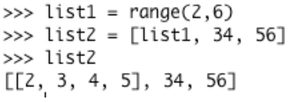


```
>>> myList = range(1977,1989)
>>> myList
[1977, 1978, 1979, 1980, 1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988]
>>> myList.append(1993)
>>> myList.append(1923)
>>> myList
[1977, 1978, 1979, 1980, 1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988, 1993, 1923]
>>> myList.sort()
>>> myList
[1923, 1977, 1978, 1979, 1980, 1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988, 1993]
>>> del myList[7]
>>> myList
[1923, 1977, 1978, 1979, 1980, 1981, 1982, 1984, 1985, 1986, 1987, 1988, 1993]
```

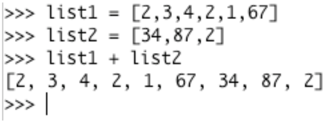


More on Lists

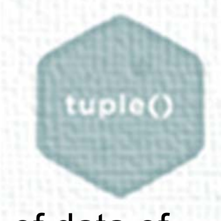
- **Nesting lists** with assigned lists is allowed, also using range() – created lists
- You can simply **add** two lists (since nesting can be challenging sometimes)



```
>>> list1 = range(2,6)
>>> list2 = [list1, 34, 56]
>>> list2
[[2, 3, 4, 5], 34, 56]
```



```
>>> list1 = [2,3,4,2,1,67]
>>> list2 = [34,87,2]
>>> list1 + list2
[2, 3, 4, 2, 1, 67, 34, 87, 2]
>>> |
```



Tuples

- Very similar to Lists (also a sequence of data of different types) but **immutable**
- Applied where it has to be ensured that the collection of values does **not change** (coordinate pairs, fixed sequence of input values)

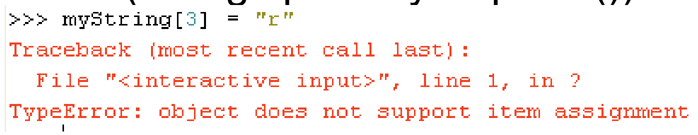
```
myTuple = (‘student1’, ‘student2’)
```

- Useful for print statements

```
age = 44
name = “Paul”
print “%s is %d years old“ %(name,age)
```

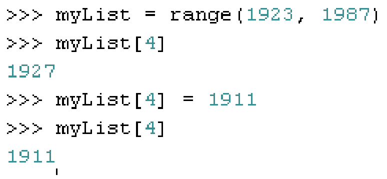
Mutable vs. Non-Mutable Sequence Types

- Tuples are immutable objects once created (Strings partially: replace())



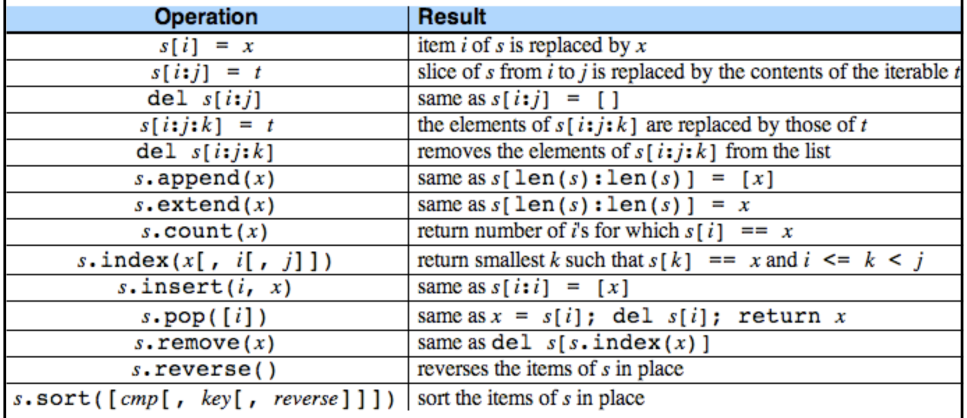
```
>>> myString[3] = "r"
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
TypeError: object does not support item assignment
```

- Lists are mutable (in-place modification, append(), extend())
- Arrays? More later.



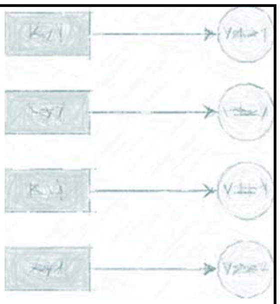
```
>>> myList = range(1923, 1987)
>>> myList[4]
1927
>>> myList[4] = 1911
>>> myList[4]
1911
```

In-place modification on Mutable Sequences

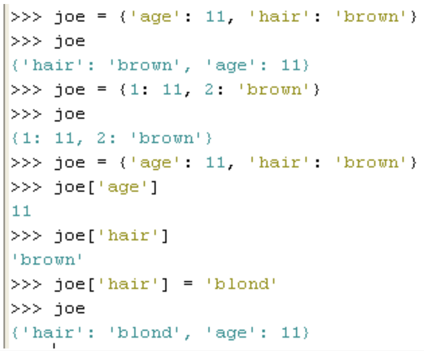


Operation	Result
s[*i*] = *x*	item *i* of *s* is replaced by *x*
s[*i*:*j*] = *t*	slice of *s* from *i* to *j* is replaced by the contents of the iterable *t*
del *s*[*i*:*j*]	same as *s*[*i*:*j*] = []
s[*i*:*j*:*k*] = *t*	the elements of *s*[*i*:*j*:*k*] are replaced by those of *t*
del *s*[*i*:*j*:*k*]	removes the elements of *s*[*i*:*j*:*k*] from the list
s.append(*x*)	same as *s*[len(*s*):len(*s*)] = [*x*]
s.extend(*x*)	same as *s*[len(*s*):len(*s*)] = *x*
s.count(*x*)	return number of *i*'s for which *s*[*i*] == *x*
s.index(*x*[, *i*[, *j*]])	return smallest *k* such that *s*[*k*] == *x* and *i* <= *k* < *j*
s.insert(*i*, *x*)	same as *s*[*i*:*i*] = [*x*]
s.pop([*i*])	same as *x* = *s*[*i*]; del *s*[*i*]; return *x*
s.remove(*x*)	same as del *s*[*s*.index(*x*)]
s.reverse()	reverses the items of *s* in place
s.sort([*cmp*[, *key*[, *reverse*]]])	sort the items of *s* in place

Mappings 1

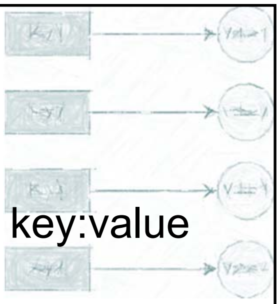


- **Dictionary (`class dict`)**
- **“Associative arrays”**:
  Entries of key-value pairs
- Inquire different “values” by their keys
  `d={key:value,…}`
- Hash table
- **Unique keys!**
- **Mutable values**
- **Immutable keys**
- Different types



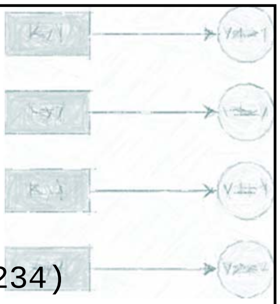
```
>>> joe = {'age': 11, 'hair': 'brown'}
>>> joe
{'hair': 'brown', 'age': 11}
>>> joe = {1: 11, 2: 'brown'}
>>> joe
{1: 11, 2: 'brown'}
>>> joe = {'age': 11, 'hair': 'brown'}
>>> joe['age']
11
>>> joe['hair']
'brown'
>>> joe['hair'] = 'blond'
>>> joe
{'hair': 'blond', 'age': 11}
```

Mappings 2



- **del** function allows to delete a key:value pair: `del myDict[“key”]`
- **Overwrites** if the same key is used a second time
- `dict.keys()` returns a **list** of keys used in dict
- `dict.has_key(‘key’)` returns a **Boolean value** (or `‘key’ in dict`)

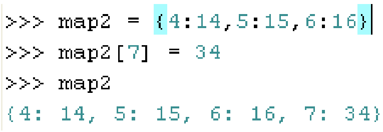
Mappings 3



- Alternative assignment:

  `myDict = dict(key1=123, key2=234)`
  `#if keys are` **`strings`**

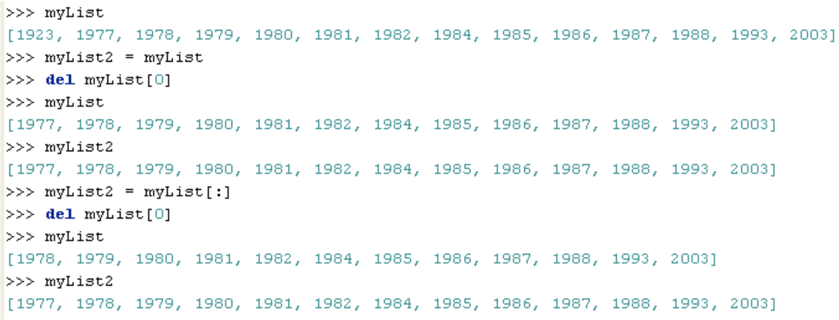
- **Updating**:
  with another dictionary: `map1.update(map2)`
- **Extending** a dictionary



```
>>> map2 = {4:14,5:15,6:16}
>>> map2[7] = 34
>>> map2
{4: 14, 5: 15, 6: 16, 7: 34}
```

Caution, References!

- Making a **copy** of a list is not the same as defining a second variable that **refers** to the same object



```
>>> myList
[1923, 1977, 1978, 1979, 1980, 1981, 1982, 1984, 1985, 1986, 1987, 1988, 1993, 2003]
>>> myList2 = myList
>>> del myList[0]
>>> myList
[1977, 1978, 1979, 1980, 1981, 1982, 1984, 1985, 1986, 1987, 1988, 1993, 2003]
>>> myList2
[1977, 1978, 1979, 1980, 1981, 1982, 1984, 1985, 1986, 1987, 1988, 1993, 2003]
>>> myList2 = myList[:]
>>> del myList[0]
>>> myList
[1978, 1979, 1980, 1981, 1982, 1984, 1985, 1986, 1987, 1988, 1993, 2003]
>>> myList2
[1977, 1978, 1979, 1980, 1981, 1982, 1984, 1985, 1986, 1987, 1988, 1993, 2003]
```

Further Data Types

- There are different **additional** types such as stacks, sets, dates, bags, queues, files, …
- **User-defined** or complex data types:

  -here, objects are used that have to be created using classes we defined

  -these objects may have **customer-defined** behavior and properties (example point object …; example list)

Note…

- That we are using already **objects** and their **methods/properties**
- Python shows a high standard of built-in (in Python library) variables, objects and functions
- User-specific data types can be created by defining classes …
- While working with the geoprocessor you already have seen how important these things are …

Summary

- Python offers different **complex data types** with **built-in** functions
- We work with them all the time: **Strings**, **Lists**! They make Python so powerful
- Working with the Geoprocessor means bringing together data **types**, **looping** and **decision** logic of Python with **Geoprocessing** functionality to work with spatial data
- “Two worlds that talk to each other!”
- Quick review on exercise class02 (extend to line)

An outlook on arcpy work

- => Tool functions (ArcToolbox)
- => Cataloguing, organizing and listing spatial data: Batching
- => Describing spatial data and their properties
- => Creating, editing and manipulating spatial data (data access)
- => More complex tasks combining above (e.g., Sampling in space w/ geometry)