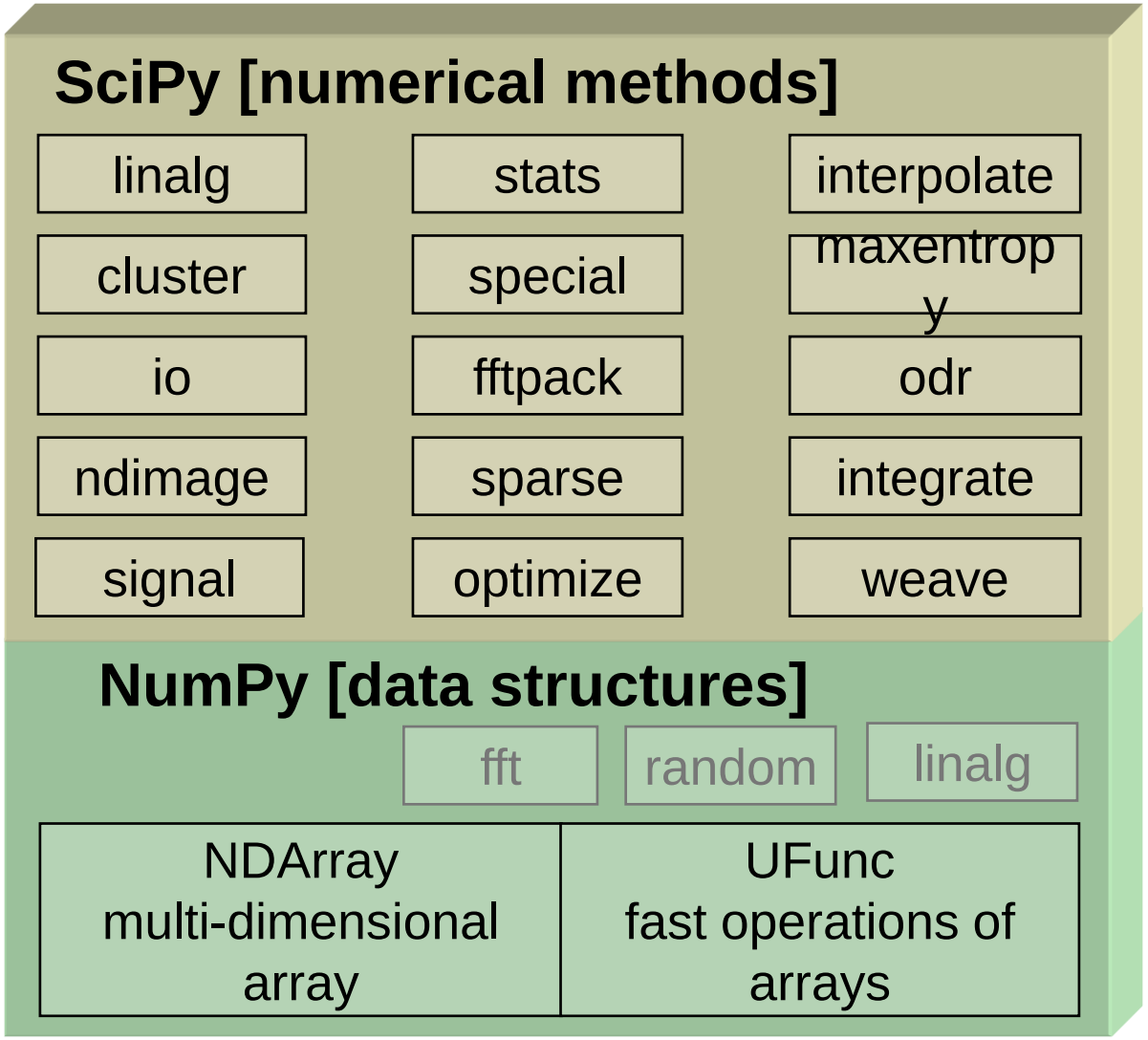
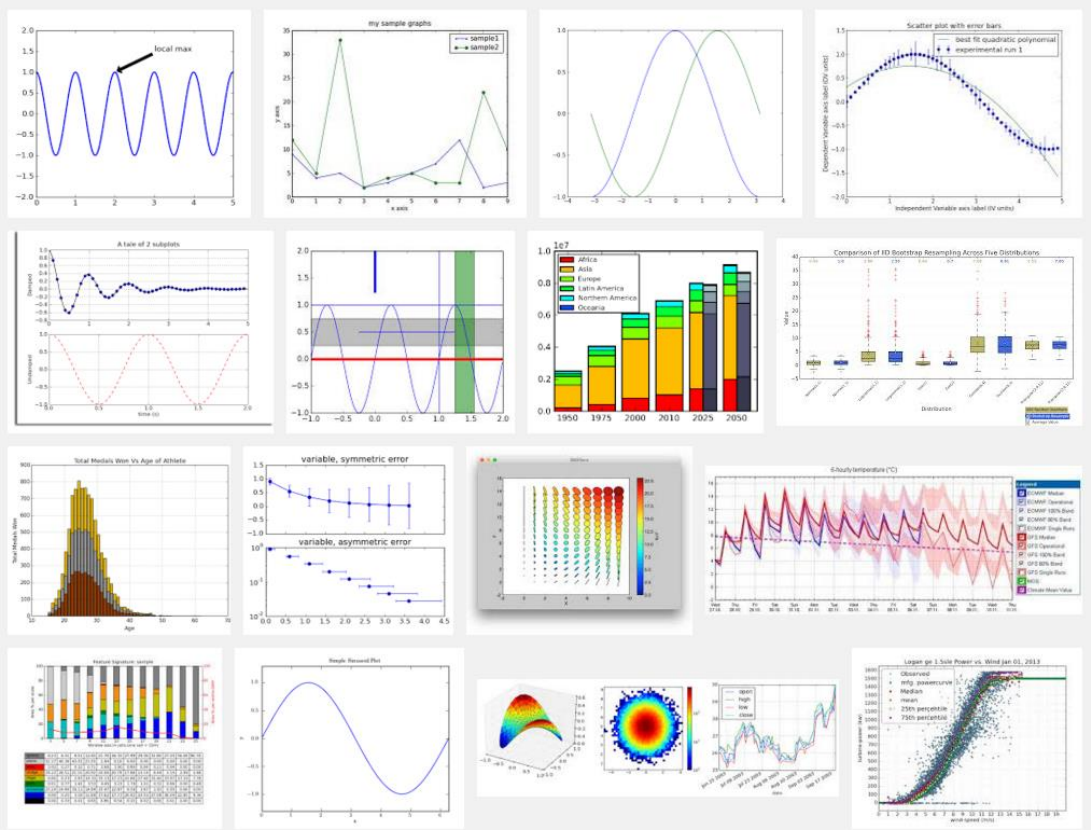
Python for Engineering

Dr Wojciech Palubicki

Great Math Libraries



MATPLOTLIB


local max
my sample graphs
sample1
sample2
Scatter plot with error bars
best fit quadratic polynomial
experimental run 1
A tale of 2 subplots
Africa
Asia
Europe
Latin America
Northern America
Oceania
1950
1975
2000
2010
2025
2050
Comparison of ID Bootstrap Resampling Across Five Distributions
Total Medals Won Vs Age of Athlete
variable, symmetric error
variable, asymmetric error
6-hourly temperature (°C)
Simple Reward Plot
Logan ge 1.5 kW Power vs. Wind Jan 01, 2013

Topics

- Interpolation
- Approximation
- Dimensionality Reduction
- Statistics
- Differential Equation Modeling

Course Work

- Multiple choice tests (50%)
- Group presentation (15%)
- Group programming assignment (35%)
- wp.faculty.wmi.amu.edu.pl/PFE.html

Programming Languages

- Microcode
- Machine code
- Assembly Language (symbolic representation of machine code)
- Low-level Programming Language (FORTRAN, COBOL, BASIC, C)
- High-level Programming Language (Java, **Python**, Prolog, MATLAB, R)

Compilation

- Compiler is a program which converts programs written in a high-level language to an equivalent low-level language program
- *Pros*:
  - Compile once, run it many times
  - The compiler can make the runtime of a program more efficient, even if its optimization might take some time
- *Cons*:
  - Debugging of code needs a large body of additional software tools

Interpreted Code

- Code that isn’t compiled is interpreted
- Python uses generated „byte-code” before interpretation (files *.**pyc**)
- *Pros*:
  - Interactive response of the computer
  - Better debugging
  - Easier to add/modify code during execution of programs
- *Cons:*
  - Slower

Python

- Python is a dynamic **interpreted** programming language
- There is no declaration of variable types, parameters, functions or methods in the source code
- All object types are defined by the interpreter during runtime

Python Interpreter

```
>>> a = 6
>>> a
6
>>> a + 2
8
>>> a = 'hello'
'hello'
>>> len(a)
5
```

Python Interpreter

```
>>> a = 6
>>> a
6
>>> a + 2
8
>>> a = 'hello'
'hello'
>>> len(a)
5
```

Start the Ipython shell

Source code

- Python source code uses the extension „.py” and such files are called **modules**
- To run a python module called ’hello.py’ you can type ‚python hello.py Wojtek’ or run it with VIDLE

```
1  import sys
2
3  print 'Witaj', sys.argv[1]
```

Conditional instructions: *if - else*

```
if imie == 'Wojtek':
    print 'witaj '+ imie
else:
    print 'kto jest ' + imie
```

Functions

- Functions are indicated with the command: **def**
- Indentation in Python determines the interpretation of instructions
  - A logical block of code should always have the same indentation

```
def funkcja(s, b):

    result = s+s+s

    if b:
        result = result + '!!!'

    return result
```

Functions

- Functions are indicated with the command: **def**
- Indentation in Python determines the interpretation of instructions
  - A logical block of code should always have the same indentation

```
def funkcja(s, b):

    result = s+s+s

if b:
        result = result + '!!!!'

    return result
```

Code interpreted during runtime

- The following code will work correctly if the argument is *Wojtek* despite the fact that the source code contains obvious errors

```
imie = sys.argv[1]

if imie != 'Wojtek':
    print funkcjaaaa(imie) + '!!!'
else:
    print funkcja(imie,1)
```

Exercises

- Write a function that greets you if you use your own name as a function argument but otherwise asks who it is

Python Modules

TAB key should expand a list of available modules (ipython only)

```
In [41]: sys.
sys.api_version            sys.float_repr_style       sys.path_hooks
sys.argv                   sys.getcheckinterval       sys.path_importer_cache
sys.base_exec_prefix       sys.getdefaultencoding     sys.platform
sys.base_prefix            sys.getfilesystemencoding  sys.prefix
sys.builtin_module_names   sys.getprivatedllspath     sys.ps1
sys.byteorder              sys.getprofile             sys.ps2
sys.call_tracing           sys.getrecursionlimit      sys.ps3
sys.callstats              sys.getrefcount            sys.py3kwarning
sys.copyright              sys.getsizeof              sys.setcheckinterval
sys.displayhook            sys.gettrace               sys.setprivatedllspath
sys.dllhandle              sys.getwindowsversion      sys.setprofile
sys.dont_write_bytecode    sys.hexversion             sys.setrecursionlimit
sys.exc_clear              sys.last_traceback         sys.settrace
sys.exc_info               sys.last_type              sys.stderr
sys.exc_type               sys.last_value             sys.stdin
sys.excepthook             sys.long_info              sys.stdout
sys.exec_prefix            sys.maxint                 sys.subversion
sys.executable             sys.maxsize                sys.version
sys.exit                   sys.maxunicode             sys.version_info
sys.exitfunc               sys.meta_path              sys.warnoptions
sys.flags                  sys.modules                sys.winver
sys.float_info             sys.path
```

Description of methods and functions

- A description of methods and functions is available using **help()**, **dir()** or **?**

```
In [44]: help(sys.exit)
Help on built-in function exit in module sys:

exit(...)
    exit([status])

    Exit the interpreter by raising SystemExit(status).
    If the status is omitted or None, it defaults to zero (i.e., success).
    If the status is an integer, it will be used as the system exit status.
    If it is another kind of object, it will be printed and the system
    exit status will be one (i.e., failure).
```

Python Strings

- Python has a class „str” providing many different string operations
- Strings can be defined using ' or ”
- Using backslash \ works as usual, e.g. \n,\\ creates a new line
- Strings are *immutable*, meaning you cannot change individual characters after a variable has been defined
- Polish symbols:
  - # -*- coding: utf-8 -*- (in the heading)
  - Use character *u* before quotes ”

String splicing

Individual characters in a string are **indexed** using [] (first index is 0, e.g. str[1] is ‚a' in the string ‚hallo')

```
In [48]: s = 'hallo'

In [49]: s[1:4]
Out[49]: 'all'

In [50]: s[1:]
Out[50]: 'allo'

In [51]: s[:]
Out[51]: 'hallo'

In [52]: s[-1]
Out[52]: 'o'

In [53]: s[:-3]
Out[53]: 'ha'
```

Exercises

- Create a function that takes *a number* as *input* and returns the *string* ‚The number of cookies is: #’ if and only if the number is smaller or equal to 9 or ‚too many cookies’ if the number is greater than 9
- Create a function that removes the first and last two characters of any string
- Create a function that takes two strings as input and returns two new strings where the first two characters are exchanged between the strings

Python Lists

- Lists are created using brackets notation **[]**

```
In [87]: liczby = [1,2,3]

In [88]: liczby[0]
Out[88]: 1

In [89]: liczby[2]
Out[89]: 3

In [90]: len(liczby)
Out[90]: 3
```

Python Lists

- Lists are created using brackets notation **[]**

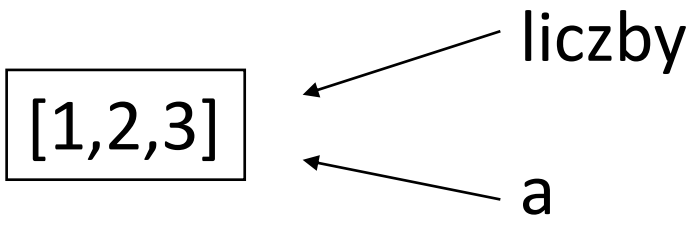
```
a = liczby
```

Does not copy a list!

```
In [87]: liczby = [1,2,3]

In [88]: liczby[0]
Out[88]: 1

In [89]: liczby[2]
Out[89]: 3

In [90]: len(liczby)
Out[90]: 3
```

Python Lists

- Lists are created using brackets notation **[]**




```
In [87]: liczby = [1,2,3]

In [88]: liczby[0]
Out[88]: 1

In [89]: liczby[2]
Out[89]: 3

In [90]: len(liczby)
Out[90]: 3
```

FOR and *IN*

- Loops in python are defined, for example, with the command:

for **var** in **list**

```
liczby = [1, 2, 3]
suma = 0
for n in liczby:
    suma += n

print suma #6
```

FOR and *IN*

- Loops in python are defined, for example, with the command:

for **var** in **list**

```
for i in range(100):
    print i
```

```
liczby = [1, 2, 3]
suma = 0
for n in liczby:
    suma += n

print suma #6
```

List Comprehensions

```
>>> mylist = [1, 2, 3]
>>> for i in mylist:
...     print(i)
1
2
3
```

```
>>> mylist = [x*x for x in range(3)]
>>> for i in mylist:
...     print(i)
0
1
4
```

- Slicing works just as in the case of strings, e.g. liczby[1:] → 2, 3

```
In [95]: list.
list.append  list.extend  list.insert  list.pop     list.reverse
list.count   list.index   list.mro     list.remove  list.sort
```

Sorting

- With the function **sorted**()

```
In [3]: a = [3,1,2,7,9]

In [4]: print sorted(a)
[1, 2, 3, 7, 9]
```

```
In [5]: a = ['a', 'b', 'ZZ', 'C']

In [6]: print sorted(a)
['C', 'ZZ', 'a', 'b']
```

Sorting with the parameter **(key=)**

```
In [9]: a = ['a', 'bbbb', 'cc', 'ddd']

In [10]: print sorted(a, key=len)
['a', 'cc', 'ddd', 'bbbb']
```

Sorting with the parameter **(key=)**

```
In [9]: a = ['a', 'bbbb', 'cc', 'ddd']

In [10]: print sorted(a, key=len)
['a', 'cc', 'ddd', 'bbbb']
```

,a’ ,bbbb’ ,cc’ ,ddd’

Sorting with the parameter **(key=)**

```
In [9]: a = ['a', 'bbbb', 'cc', 'ddd']

In [10]: print sorted(a, key=len)
['a', 'cc', 'ddd', 'bbbb']
```

‚a’	‚bbbb’	‚cc’	‚ddd’
↓	↓	↓	↓
1	4	2	3

Sorting with the parameter **(key=)**

```
In [9]: a = ['a', 'bbbb', 'cc', 'ddd']

In [10]: print sorted(a, key=len)
['a', 'cc', 'ddd', 'bbbb']
```

	‚a’	‚bbbb’	‚cc’	‚ddd’
	1	4	2	3
sorting →	1	2	3	4
	‚a’	‚cc’	‚ddd’	‚bbbb’

Sorting using the **sort()** function

```
In [26]: a = ['a', 'bbbb', 'cc', 'ddd']

In [27]: a.sort(key=len)

In [28]: print a
['a', 'cc', 'ddd', 'bbbb']
```

Sorting using the **sort()** function

```
In [26]: a = ['a', 'bbbb', 'cc', 'ddd']

In [27]: a.sort(key=len)

In [28]: print a
['a', 'cc', 'ddd', 'bbbb']
```

```
In [29]: print a.sort(key=len)
None
```

Tuples

- Tuples are grouping of elements of a given length(e.g. a point in a 2D Cartesian coordinate system)
- A tuple is immutable and cannot change in size in contrast to lists
- Tuples are defined using brackets **()**

```
In [11]: a = (1,2,'a')

In [12]: a
Out[12]: (1, 2, 'a')
```

Tuples vs. Lists

```
>>> def a():
...     x=[1,2,3,4,5]
...     y=x[2]
...
>>> def b():
...     x=(1,2,3,4,5)
...     y=x[2]
...
>>> import dis
>>> dis.dis(a)
  2           0 LOAD_CONST               1 (1)
              3 LOAD_CONST               2 (2)
              6 LOAD_CONST               3 (3)
              9 LOAD_CONST               4 (4)
             12 LOAD_CONST               5 (5)
             15 BUILD_LIST               5
             18 STORE_FAST               0 (x)

  3          21 LOAD_FAST                0 (x)
             24 LOAD_CONST               2 (2)
             27 BINARY_SUBSCR
             28 STORE_FAST               1 (y)
             31 LOAD_CONST               0 (None)
             34 RETURN_VALUE
>>> dis.dis(b)
  2           0 LOAD_CONST               6 ((1, 2, 3, 4, 5))
              3 STORE_FAST               0 (x)

  3           6 LOAD_FAST                0 (x)
              9 LOAD_CONST               2 (2)
             12 BINARY_SUBSCR
             13 STORE_FAST               1 (y)
             16 LOAD_CONST               0 (None)
             19 RETURN_VALUE
```

Exercises

- For a given list of strings write a function that returns the number of strings in that list where the length is greater than 2 and the last character is identical to the first
- For a given list of strings write a function that returns a sorted list with all strings starting with the character x at the beginning (define two lists)
- For a given list of numbers return a list where all identical, neighboring numbers are contracted to a single number (e.g. [1,2,2,3]→ [1,2,3])

Dictionary

- Dictionaries are defined using {}

```
In [30]: dict = {}

In [31]: dict['r'] = 'raz'

In [32]: dict['d'] = 'dwa'

In [33]: dict['t'] = 'trzy'
```

Dictionary

- Dictionaries are defined using {}

```
In [30]: dict = {}

In [31]: dict['r'] = 'raz'

In [32]: dict['d'] = 'dwa'

In [33]: dict['t'] = 'trzy'
```

```
In [34]: print dict['r']
raz

In [35]: dict['r']  = 1
```

Dictionary

- Dictionaries are defined using {}

```
In [30]: dict = {}

In [31]: dict['r'] = 'raz'

In [32]: dict['d'] = 'dwa'

In [33]: dict['t'] = 'trzy'
```

```
In [34]: print dict['r']
raz

In [35]: dict['r']  = 1
```

```
In [38]: dict={'r':'raz', 'd':'dwa', 't':'trzy'}

In [39]: print dict
{'r': 'raz', 'd': 'dwa', 't': 'trzy'}
```

Dictionary

Keys	Values
‚r’	‚raz’
‚d’	‚dwa’
‚t’	‚trzy’

dict

Dictionary

Keys		Values
‚r’	→	‚raz’
‚d’	→	‚dwa’
‚t’	→	‚trzy’

dict

```
In [41]: print dict.keys()
['r', 'd', 't']
```

```
In [42]: print dict.values()
['raz', 'dwa', 'trzy']
```

```
In [40]: for key in dict: print key
r
d
t
```

```
In [43]:  for key in sorted(dict.keys()):
   ....:          print key, dict[key]
   ....:
d dwa
r raz
t trzy
```

```
In [40]: for key in dict: print key
r
d
t
```

```
In [43]:  for key in sorted(dict.keys()):
   ....:          print key, dict[key]
   ....:
d dwa
r raz
t trzy
```

```
In [56]: 'r' in dict
Out[56]: True
```

```
In [40]: for key in dict: print key
r
d
t
```

```
In [43]:  for key in sorted(dict.keys()):
   ....:          print key, dict[key]
   ....:
d dwa
r raz
t trzy
```

```
In [56]: 'r' in dict
Out[56]: True
```

```
In [60]: 'raz' in dict.values()
Out[60]: True
```

Del

- Operator ‚del’ deletes elements

```
In [48]: dict = {'a':1, 'b':2, 'c':3}

In [49]: del dict['b']

In [50]: print dict
{'a': 1, 'c': 3}
```

Files

```
In [51]: f = open('hallo.py', 'rU')

In [52]: for line in f: print line,

In [53]: f.close()
```

'r' for reading, 'w' for writing, and 'a' for appending

Files

```
In [51]: f = open('hallo.py', 'rU')

In [52]: for line in f: print line,

In [53]: f.close()
```

```
In [67]: f.
f.close       f.fileno      f.name        f.readinto    f.softspace   f.writelines
f.closed      f.flush       f.newlines    f.readline    f.tell        f.xreadlines
f.encoding    f.isatty      f.next        f.readlines   f.truncate
f.errors      f.mode        f.read        f.seek        f.write
```

Exercises

- Write a program that opens a file containing two columns and save it as a dictionary where the values of the first column are the keys for values in the second column
- Write a program that counts the number of words in the file and returns how many times each word is repeated

The function **split**() from the module **str** creates a list of strings from a string that contains separated strings