Introduction to python 3

Kevin Sturm and Winfried Auzinger

Outline

1. The basics
2. SciPy
3. Plotting with matplotlib
4. Sympy

Python references

- Good python book *Python 3 (2017 edition)* by Johannes Ernesti and Peter Kaiser
- online documentation: `https://docs.python.org/3.6/`

Historical facts

- developed in the nineties by Guido van Rossum in Amsterdam at Centrum voor Wiskunde en Informatica
- the name "python" comes from the comedy "*Monty Python*"
- python **version 3.0** was released in December 2008
- one of the most popular programming languages
- designed for *functional* and *object oriented* programming
- programs that partially use python:
  - Google Mail
  - Google Maps
  - YouTube
  - Dropbox
  - reddit
  - Battlefield 2
  - BitTorrent

Why python?

What does python offer?

- Interactive
- Interpreted
- Modular
- Object-oriented
- Portable
- High level
- Extensible in C++ & C

Why is python good for scientifc computing?

- open source / free
- many libraries, e.g.,
- scientific computing: numpy, scipy
- symbolic math: sympy
- plotting: matplotlib
- excellent PDE solver software: ngsolve, FEniCs, Firedrake, ...

How to start python?

- Python can either be used **interactively**: simply type "python3" or "ipython3" (to start IPython) into the shell
- we can also **execute python code** written in a file "file.py" by typing "python3 file.py" into the shell

Let's start with a hello world example:

Listing 1: hello_world.py

```python
""" This is our first program """

print("Hello world!")
```

Float

declaration of floats

```
>>> x = 987.27
>>> x
987.27
```

division

```
>>> y = 2.27
>>> x/y
434.92070484581495
```

floor division

```
>>> x//y
434.0
```

addition and subtraction

```
>>> x = 987.27
>>> y = 2.0
>>> x+y
989.27
>>> x-y
985.27
```

powers

```
>>> x**2
974702.0529
>>> x**3
962294095.766583
>>> x**0.5 # square root
31.4208529483208
```

multiplication

```
>>> x*y
1974.54
>>> x*-y
```

Integers

calculator

```
>>> 1+3
4
>>> 3-10
-7
>>> 30*3
90
```

declaration of integer

```
>>> x = 987
>>> x
987
>>> z = int(10.0)
>>> z
10
```

multiplication and division

```
>>> y = 2
>>> x/y
493.5
>>> 5/3
1.6666666666666667
```

floor division

```
>>> x//y
493
```

conversion of float to integer

```
>>> x = 1.4
>>> y = int(x)
>>> y
1
>>> x + 3
4.4
```

- remember: float + int = float

Complex number

- imaginary unit in python is j
- recall $(a + ib) * (c + id) := ac - db + i(bc + ad)$

```
>>> z = 1.0 + 5j  # complex number with real 1 and imag 5
>>> z.conjugate()  # conjugate complex number
(1-5j)

>>> z = complex(1,5) # equivalent to 1+5j

>>> z.imag # return imaginary part
5.0
>>> z.real # return real part
1.0
```

Complex number (continued)

multiplication of complex numbers

```
>>> z1 = 1 + 4j
>>> z2 = 2 - 4j

>>> z1*z2  # multiply z1 and z2
(18+4j)

>>> # Let us verify this is correct

>>> a, b, c, d  = z1.real, z1.imag, z2.real, z2.imag

>>> a*c - b*d
18.0
>>> b*c + a*d
4.0
```

Strings

declaration of strings

```
>>> a = "hello" # assign hello
>>> a
'hello'
```

addition of strings

```
>>> a+a
'hellohello'
>>> a+" cool"
'hello cool'
```

referencing letters

```
>>> fourth =   a[3]  # 4th letter
>>> fourth
'l'
>>> last = a[-1]  # last letter
>>> last
'o'
```

conversion of float and integer to string

```
>>> x = 987.27
>>> s1 = str(x)
>>> s1
'987.27'
>>> n = 10
>>> s2 = str(n)
>>> s2
'10'
```

Strings (continued)

lower and upper case

```
>>> a = "hello" # assign hello
>>> a.upper()
'HELLO'

>>> a = "HELLO"
>>> a.lower()
'hello'
>>> a
'HELLO'

>>> a = "Hello"
>>> a.swapcase()
'hELLO'
>>> a
'Hello'
```

accessing letters

```
>>> s = "This is a long sentence!"
>>> s[::3] # every third letter
'Tss nstc'
>>> s = "z"
>>> 10*s
'zzzzzzzzzz'
```

Splitting and concatenation

```
>>> name = "This is a long sentence."
>>> name.split()
['This', 'is', 'a', 'long', 'sentence.']
>>> name
'This is a long sentence.'
```

inserting strings

```
>>> 'Insert here: {}'.format('Inserted string')
'Insert here: Inserted string'
```

Lists

declaration of list

```
>>> l = []   # empty list
>>> l
[]
>>> l = [1, 2, 3] # integers list
>>> l
[1, 2, 3]
>>> l = [1.0, 3.0, 3,0] # float list
```

lists can contain anything

```
>>> l1 = [1,2,3]
>>> l2 = ["hello", [], "new"]
>>> l = [l1, l2]
>>> l
[[1, 2, 3], ['hello', [], 'new']]
```

other ways to generate lists

```
>>> l1 = [1]*5
>>> l1
[1, 1, 1, 1, 1]
>>> l2 = [k for k in range(5)]
>>> l2
[0, 1, 2, 3, 4]
```

The last command is similar to the mathematical definition $\{k : \ k = 0, 1, 2, 3, 4\}$.
addition of lists

```
>>> l1 = [1,2,3]
>>> l2 = [4,5,6]
>>> l1+l2
[1, 2, 3, 4, 5, 6]
```

multiplication of lists is not supported!!

More on lists

The *list* class has the following methods:

- append
- clear
- copy
- count
- extend
- index
- insert
- pop
- remove
- reverse
- sort

```
>>> l = [1, 2, 3, 4, 4]
>>> l
[1, 2, 3, 4, 4]

>>> l.reverse()
>>> l
[4, 4, 3, 2, 1]

>>> l.pop(3)
2
>>> l
[4, 4, 3, 1]

>>> # print every 2nd element
>>> # start with index 1
>>> # go until end of list -1
>>> # the : operation is called slicing
>>> l[1:-1:2]
[4]
```

Tuple

- Tuple are essentially uneditable lists. We use round parenthesis.
- referencing possible, but no assignment
- to be used when list should not be modified

declaration of list

```
>>> l = ()  # empty tuple
>>> l
()
>>> l = (1, 2, 3) # tuple of integers
>>> l
(1, 2, 3)
>>> l = tuple([1.0, 3.0, 3,0]) # conversion of list to tuple
>>> l
(1.0, 3.0, 3, 0)
```

adding tuples

```
>>> l+l
(1.0, 3.0, 3, 0, 1.0, 3.0, 3, 0)
>>> 4*l
(1.0, 3.0, 3, 0, 1.0, 3.0, 3, 0, 1.0, 3.0, 3, 0, 1.0, 3.0, 3, 0)
```

Bool and logical operators

bool True or False

```
>>> t = True
>>> t
True
>>> f = False
>>> f
False
>>> f == t
False
```

"and", "or", and "not

```
>>> t and f
False
>>> t or f
True
>>> not f == t
True
```

Possibilities for "or":

x	y	x or y
True	True	True
True	False	True
False	True	True
False	False	False

Possibilities for "and":

x	y	x and y
True	True	True
True	False	False
False	True	False
False	False	False

If-else

simple if-else statement

Listing 2: if_else.py

```
if condition:
    command
else:
    another command
```

When we have more than one condition we use *elif*:

Listing 3: if_else2.py

```
if condition1:
    first command
elif condition2:
    second command
else:
    third command
```

If-else example

Listing 4: if_else_ex.py

```python
if x == 1:
    print("x has value 1")
elif x == 2:
    print("x has value 2")
```

Listing 5: if_else_ex2.py

```python
if x == 1:
    print("x has value 1")
else:
    print("x has another value")
```

for loop

Listing 6: for_loop.py

```python
for n in range(10):
    print(n)
```

- Here *n* ranges from 0 to 9 and is printed after each loop.
- general syntax is *range(start, stop, steps)*
- *start* and *steps* are optional

Listing 7: for_loop2.py

```python
l = [0, 1, 'hello', True, False]

for n in l:
    print(n)
```

for loop (continued)

- use *enumerate* to count the element in the loop

Listing 8: for_loop_en.py

```
l = ['one', 'two', 'three', 'four', 'five']

for n, s in enumerate(l):
    print('Item number ', n, ' item itself ', s)
```

While loop

The syntax of a python while loop is as follows.

```
while statement:
    do stuff
```

- "do stuff" is executed as long as statement is true.
- notice again the indention!
- use `break` to leave a while loop
- use `continue` to go to the next loop

Listing 9: while_loop.py

```python
counter = 10

while counter > 0:
    print("counter is", counter)
    counter -= 1
```

Functions

Let's have a look at an example function.

Listing 10: func.py

```
def my_func(x):
    x = x + 1.0
    return x
```

- indention in python replaces brackets!!!
- a function always starts with *def*
- a *return* is not mandatory
- without *return* the function returns None.

Functions (continued)

- anonymous functions can be defined using *lambda* keyword

```
>>> f = lambda x: x**2 # define lambda function f
>>> f(2)
4
```

a more complicated example

```
>>> f = lambda x: x**2 if x < 0 else x**3
>>> f(2)
8
```

Listing 11: lambda_func.py

```
def f(x):
    if x < 0:
        return x**2
    else:
        return x**3
```

Functions (optional arguments)

- It is possible to give functions optional arguments.

Listing 12: func_opt.py

```
def f(x, y=None):

    if y == None:
        return x**2
    else:
        return x**2 + y**2
print(f(1))
print(f(1,2))
```

Dictionaries

- make a dictionary with {} and : to signify a *key* and a *value*

```
>>> value1 = 1.0
>>> value2 = 2.0
>>> my_dict = {'key1':value1,'key2':value2}

>>> print(my_dict)
{'key1': 1.0, 'key2': 2.0}

>>> my_dict['key1'] # access value1
1.0

>>> 'key2' in my_dict
True
```

Dictionaries (continued)

Accessing the values and the keys

```
>>> # Make a dictionary with {} and : to signify a key and a value
>>> value1 = 1.0
>>> value2 = 2.0
>>> my_dict = {'key1':value1,'key2':value2}


>>> print(my_dict.values())  # return values of dictionary
dict_values([1.0, 2.0])

>>> print(my_dict.items()) # return items
dict_items([('key1', 1.0), ('key2', 2.0)])

>>> print(my_dict.keys()) # return keys
dict_keys(['key1', 'key2'])
```

Sets

- sets are unordered lists

declaration of sets

```
>>> S = set([1,2,3,4])  # def. a set S
>>> S
{1, 2, 3, 4}

>>> S = {1,2,3,4} # equiv. definition
>>> S
{1, 2, 3, 4}
```

union $\cup$ and subtraction $\setminus$ of sets

```
>>> S1 = {1,2,3}
>>> S2 = {2,3,4}

>>> S1 - S2  # subtract S1 from S2
{1}
>>> S2 - S1  # subtract S2 from S1
{4}

>>> S1 | S2  # union of S1 and  S2
{1, 2, 3, 4}
```

Sets (continued)

alternative definition

```
>>> S1 = {2,3,4,5}
>>> S2 = {1,2,3,4}

>>> S1.intersection(S2)
{2, 3, 4}

>>> S2.union(S1)
{1, 2, 3, 4, 5}

>>> S1.difference(S2)
{5}
```

union ∪ and subtraction \ of sets

```
>>> S1 = set([1,2,3])
>>> S2 = set([2,3,4])

>>> S1 - S2   # S1/S2
{1}
>>> S2 - S1   # S2/S1
{4}

>>> S1 | S2 # union of S1 and S2
{1, 2, 3, 4}
```

adding and deleting elements

```
>>> S1.add(10) # add 10 to list
>>> S1
{10, 1, 2, 3}

>>> S1.discard(10) # remove element 10
>>> S1
{1, 2, 3}
```

Python key words

- We already know a few python key words.
- The *keywords* are part of the python programming language.
- you cannot use these names for variables or functions

and	def	finally	in	or	while
as	del	for	is	pass	with
assert	elif	from	lambda	raise	yield
break	else	global	None	return	
class	except	if	nonlocal	True	
continue	False	import	not	try	

Figure: List of python keywords

Importing modules

- import a module with command `import module_name`
- a function **func** in **module_name** can be accessed by `module_name.func`
- including with different name use `import module_name as mn`
- import specific function: `from module_name import func`
- import everything with `from module_name import *`

Math modul

Let us consider as an example the *math* package.

```
>>> import math  # import math module and use name "math"
>>> math.pi
3.141592653589793
>>> del(math) # remove math package

>>> import math as m # import math module with name "m"
>>> m.pi
3.141592653589793
>>> del(m)

>>> from math import pi # import constant pi from math
>>> pi
3.141592653589793

>>> from math import pi as pipi # import constant pi from math with name "pipi"
>>> pipi
3.141592653589793
```

Immutable vs mutable datatypes

- Python distinguishes two datatypes: mutable and immutable.
- immutable: float, int, string, tuple
- mutable: set, list, dict

The build-in function `id(variable)` shows the unique identity of a python object.

```
>>> s1 = "CompMath"
>>> s2 = "CompMath"

>>> id(s1)
139999031155696
>>> id(s2)
139999031155696

>>> s1 is s2  # check if s1 is s2
True

>>> s1 == s2  # check if s1 has same values as s2
True
```

Immutable vs mutable datatypes (continued)

Let us now check lists.

```
>>> l1 = [0.1, "CompMath"]
>>> l2 = [0.1, "CompMath"]

>>> id(l1)
139999033935048
>>> id(l2)
139999033935752

>>> l1 is l2  # check if l1 is l2
False

>>> l1 == l2  # check if l1 has same values as l2
True
```

So both lists are different, but have exactly the same values.

Immutable vs mutable datatypes (continued)

```
>>> l1 = [0.1, "CompMath"]
>>> l2 = l1

>>> l1 is l2  # check if s1 is s2
True

>>> l1 == l2  # check if s1 has same values as s2
True

>>> id(l1)
139999033935240
>>> id(l2)
139999033935240

>>> l1[0] = 0.0
>>> l1
[0.0, 'CompMath']
>>> l2
[0.0, 'CompMath']
```

- So l1 and l2 share the same reference. Changing l1 also changes l2.

Immutable vs mutable datatypes (continued)

So how can we copy a list?

```
>>> l1 = [0.1, "CompMath"]
>>> l2 = l1[:]  # this generates a copy of l1

>>> l1 is l2  # check if s1 is s2
False
>>> l1 == l2  # check if s1 has same values as s2
True

>>> id(l1)
139999033935752
>>> id(l2)
139999033935048
```

Immutable vs mutable datatypes (continued)

- if list elements are mutable itself the previous copying does not work as one might expect

```
>>> change = [0, 0, 0]
>>> l1 = [1, 2, change]
>>> l2 = l1[:]   # change is not copied here
```

In this case one can use deepcopy of the module copy.

```
>>> change = [0, 0, 0]
>>> l1 = [1, 2, change]
>>> import copy
>>> l2 = copy.deepcopy(l1)
```

Local vs global variables

How to figure our which variables are defined so far?

- `dir()` - list defined variables in scope
- `globals()` - dict of global variables
- `locals()` - dict of local variables in scope (including values)

Local vs global variables - example

Listing 13: dirs.py

```
b = 0.

def f(x):
    a = 0.0

    print("local variables in f", locals())
    print("local variables f", dir())

    return x

print("local variables in current scope", locals())

print(f(0.1))
```

Classes

Listing 14: class_ex.py

```
class simple:
    pass
```

- keyword `class` defines a class with name simple
- keyword `pass` means that the class simple does nothing

Classes

Listing 15: class_ex2.py

```
class simple_two:
    a = 0.1
    s = "hello"

t = simple_two() # define class instance

print(t.a) # print variable a
```

- keyword `class` defines a class with name simple
- keyword `pass` means that the class simple does nothing

Classes - constructor

Listing 16: class_construct.py

```
class test:

    def __init__(self, a = 0.0): # constructor
        self.a = a

C1 = test(0.1) # create instance C1 with value a = 0.1
C2 = test() # create instance C2 with default value

print(C1.a) # print value of variable a
```

- a class constructor is defined by `__init__`, which is called upon initialisation of the class
- the class test has an optimal argument *a*, which is by default 0.0

Classes - methods

Listing 17: class_method.py

```
class test:

    def __init__(self):
        print("This is the constructor.")

    def func(self):
        print("This is the func.")

C = test() # create instance C
C.func() # call func()
```

- the first argument of a method (here func(self)) must be self
- function is accessed via `C.func()`

Classes - methods

Listing 18: class_method2.py

```python
class test:

    def __init__(self):
        print("This is the constructor.")

    def func2(self, b):
        print("This is func2 with b = {}".format(b))

C = test()
C.func2(0.3) # call func2(0.3)
```

- the first argument of a method (here func(self)) must be self (see next slide)

What is self?

- self is basically a reference to the class instance
- the name does not have to be "self", but it is recommended
- the first argument of a method in a class is always self

Listing 19: self.py

```
class test():

    def __init__(self):
        print("This is the constructor.")

    def we_call_self(self):
        print("This is self", self)

C = test()
C.we_call_self()
print("This is C", C)
```

Inheriting classes

As in C++ we can inherit classes. The basic syntax is as follows:

```
class Derived_ClassName(Base_ClassName):
    statement-1
    .
    .
    .
    statement-N
```

Inheriting classes: example

Listing 20: inherit.py

```
class Base_Class():

    def f(self, x):
        return x

class Derived_Class(Base_Class):

    def g(self, x, y):
        return x + y
```

- `Base_Class()` contains the functions `f(x)`
- `Derived_Class` extends `Base_Class()` by `g(x, y)`

Reading files

- we can read a file with `open("filename", 'r')`

We now want to read the file

Listing 21: readme.txt

```
This is CompMath.

We want to read this file.
```

```
>>> file = open("code/code_lec2/readme.txt", 'r')
>>> print(file.readlines())
['This is CompMath.\n', '\n', 'We want to read this file.\n']
>>> file.close()
```

Writing to files

- we can write to a file with `open("filename", 'w')`
- if "filename" is not there it will be created

```
file = open("code/code_lec2/writeme.txt", 'w+')
file.write("We write this into writeme.txt")
file.close()
```

Further options of `open()`

The function open has the following options. (Taken from `help(open)`).

'r'	open for reading (default)
'w'	open for writing, truncating the file first
'x'	create a new file and open it for writing
'a'	open for writing, appending to the end of the file if it exists
'b'	binary mode
't'	text mode (default)
'+'	open a disk file for updating (reading and writing)
'U'	universal newline mode (deprecated)

Reading and writing lines

Now suppose we want to add text to the beginning of the file *prepend.txt*

```
file = open("prepend.txt", 'a+')  # open file prepend.txt
file.seek(0)  # start at beginning of file
s = ["This text should go at the beginning."]
file.writelines(s)
file.close()
```

Doc-Strings

What is a doc string?

- doc-string is convenient way do describe document modules, functions, classes, and methods.

How do we define a doc string?

- a doc-string has the syntax `""" documentation here """`

How do we use a doc string?

- The doc string can be accessed with `.__doc__`.

Doc-String: example

Listing 22: doc_string.py

```
""" This is a doc string. """

def f(x, y = 0.0):
    """
    This function adds numbers x and y.
    The variable y is optional. Default is y = 0.0
    """
    return x + y

print("call doc string with f.__doc__:", f.__doc__)
print("alternatively use help(f):", help(f))
```

Decorators

The basic decorator code structure is as follows:

```
def decor(func):
    def inner():
        func()
    return inner
```

Usage:

```
dec = decor(func)
```

- `decor` is a wrapper function - essentially a function that returns a function
- the decorator gets as argument a function (`func()`) and returns another function (`inner()`)
- the "actual" coding happens inside the `inner` function

Decorators - Example 1

Listing 23: decorator_.py

```
from math import exp

def f(x, y):
    return exp(x*y) + y

def deco(func):
    y = 0.0 # define value for y
    def f1(x):
        return func(x, y)
    return f1
```

Decorators - Example 2

Listing 24: decorator2_.py

```
from math import exp

def f(x, y):
    return exp(x*y) + y

def deco(func, y): # decorator has y as argument
    def f1(x):
        return func(x, y)
    return f1

de = deco(f, 5)

print(de(0.1))
```

Decorators - Example 3

Listing 25: decorator3_.py

```
from math import sin, cos

def func_comp(fun1, fun2):
    def f1(x):
        return fun1(fun2(x))
    return f1

de = func_comp(cos, sin)

print(de(0.1))
```

Recursion without loops

Suppose we want to implement the factorial `n!`. A loop approach would be as follows:

Listing 26: factorial_loop.py

```
def fac(n):
    val = 1
    for k in range(1, n+1):
        val = val*k

    return val

print(fac(10))

## compare with math function factorial
import math

print(math.factorial(10))
```

Recursion without loops

As second approach without loops is

Listing 27: factorial_loop_free.py

```
def fac(n):
    if n == 1:
        return 1
    else:
        return n*fac(n-1) ## function fac called with n-1

print(fac(10))
```

Recursion without loops

- using second approach avoid calling function multiple times!! Consider

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{1}{x_n}\right).$$

Listing 28: babylon_bad.py

```
def babylon(n):
    x0 = 10
    if n == 1:
        return x0
    else:
        return (1/2)*(babylon(n-1) + 2/babylon(n-1))
```

problem: if a_n is number of function calls, then $a_n = 2a_{n-1}$and hence $a_n = 2^n$ function calls are need. In total to compute recursion at stage n we need $\sum_{\ell=0}^{n} a_\ell = 2^{n+1} - 1$.

Recursion without loops

- using second approach avoid calling function multiple times!! Consider

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{1}{x_n}\right).$$

Listing 29: babylon_good.py

```
def babylon(n):
    x0 = 10
    if n == 1:
        return x0
    else:
        xn = babylon(n-1)
        return (1/2)*(xn + 2/xn)
```

better: here we have $a_n = a_{n-1}$, so $a_n = a_0 = 1$ and hence in total $\sum_{\ell=0}^{n} a_\ell = n+1$.

$*args$ and $**kwargs$

- sometimes the number of arguments a function gets is unknown. Then we can use *arg and **kwargs.
- The actual names args and kwargs are irrelevant, we could also use $*va$, only the star $*$ matters; same for kwargs.

```
def f(farg, *args, **kwarg):
    # do something with args, farg and kwarg
```

$*args$ and $**kwargs$- example

Listing 30: args.py

```
def polynom(x, *args):

    n = len(args)
    val = 0.0

    print(type(args))
    for k in range(n):
        val += args[k]*x**k

    return val

a = (1, 2, 3, 4)
print(polynom(0.1, *a))
print(polynom(0.1, 1, 2, 3, 4))
```

Measuring time

- to measure time we can use the time module

Listing 31: measuring_time.py

```
import time # time module

def tic(): # start measuring time
    global start
    start = time.time()

def toc(): # end measuring time
    if 'start' in globals():
        print("time: {}.".format(str(time.time()-start)))
    else:
        print("toc(): start time not set")
```

Evaluating functions at multiple values

- How to evaluate a function `f(x)` for a list of values, say, `l = [1, 2, 3, 4, 4, 4]`?
- solution: use `map(f, l)`

```
>>> f = lambda x: x**4
>>> l = [1, 2, 3, 4, 4, 4]
>>> map(f, l)
<map object at 0x7f541084beb8>
>>> print(list(map(f,l)))
[1, 16, 81, 256, 256, 256]
```