Introduction to Python

Fermín Huarte Larrañaga [1] Ramon Crehuet Simon [2]

[1] fermin.huarte@ub.edu
Departament de Ciència de Materials i Química Física & IQTCUB
Universitat de Barcelona

[2] ramon.crehuet@iqac.csic.es
Institut de Química Avançada de Catalunya, CSIC

May 30, 2019

Table of contents

(some)online Python Resources

- Dive into Python by Mark Pilgrim
- Python Tutor, useful for variables and flow control
- Learn Python Online
- Python from Begginers
- Sofware Carpentry
- Python 3 documentation.

Variables

- As scientists we are used to symbols containing values.
- It is extremely useful to employ variables in a program: better readability, error control, and usability!

Python source code

```
a = 6.71
b = 0.0564
n = 0.5
V = 5
T = 298.15
p = n*0.0821*T / ( V - n*b ) - n**2*a / V**2
print(p)
```

Writing/Running our first Python program

- Invoke the Python interpreter or the IPython shell
- Type the code line by line and press `return`
- A Python program is written in plain text.
- Open a simple text editor such as Vi, Gedit, Emacs.
  ALTERNATIVE: Launch `jupyter lab`
- Type the code in a notebook cell (or save it to a file **with .py extension**, for instance: `vdw.py`)
- Run it!

Basic syntax

- Programs must comply with computer grammar rules, no missprints!
- One statement per line, except when separated by semicolon:
  `a = 3 ; b = 2 ; c = (a + b)/2 ; print (c)`
- Case sensitivity
- Generally the first encountered error causes program stop, easier debugging (?)

Basic syntax

- Blank spaces **may or may not** be important
  - These statements are equivalent:

```
T=298.15
T = 298.15
T= 298.15
```

  - Instead, here blank spaces are important! (*we will see these instructions in future sessions*)

```
for i in range(5):
    j = j + i #correct, indented block

for i in range(5):
j = j + i       #incorrect syntax
```

Variables: declaration, type and assignation

- Variables need not to be declared! `Fortran`
- Data do however have a type, that is transfered to the variable upon assignation

Variable types

```
a = 6.71
type(a)
a = 6
type(-6)
a = 'basis set'
type(a)
a/2
```

Names of variables

- Use names that give an idea of the purpose of the variable
- Names may contain: letters a-z, A-Z, and underscore character "_". They may also have numbers 0-9 (not as initial character!)
- Case sensitivity
- Forbidden variable names: `and`, `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`

Numerical data types

- Integer numbers

```
a = 3
b = -2
```

- Floating point (decimal) numbers

```
pi = 3.1415927
planck = 6.626E-34 #scientific notation!
```

- Complex numbers

```
c = 3 + j5
```

Variables

- Everything in Python is an *object*.
- Variables are references to objects (actually, *instances* of an object)

```
y = 5   # y refers to an int object
x = 9.0  # x refers to an float object
d = 9/y  # d refers to an int/int => int object
i = x/5 # i refers to a float/int => float object
t = "Hello World!" # t is a string object
```

Variables

- Object types can be converted

```
y = 9 # y is int 9
i = float(y) # i is float 9.0
c = 5.9 # c is float 5.9
d = int(c) # d is int 5
d = round(c) # d is int 6.0
t = str(c) # t is str "5.9"
line = '-3.5' # line is str
e = float(line) # e is float -3.5
```

Complex numbers

- Complex numbers are supported by Python.
- Complex number $5 + 2i$ is coded `5 + 2j` in Python

```
c = 5 + 2j
a = complex(3, 1)
d = c + a
print(d.real)
print(d.imag)
```

Assignement

- This is an assignment statement: `T = 25 #degrees`
- Evaluate right-hand side, assign to left-hand side:
  `T = T + 273.15 #Kelvin`
- Long notation and short notation

```
i = i  + 1
i += 1 #this is equivalent
j = 5 * j
j *= 5 #this is equivalent
```

- Assignment is not mathematical equality!

Arithmetic Expressions

$$p = \frac{nRT}{(V - nb)} - \left(\frac{n^2 a}{V^2}\right)$$

```
p = n*0.0821*T / ( V - n*b ) - n**2*a / V**2
```

- Notice exponentiation: `**`
- Priority of operations:
  1. `**`
  2. `*`,`/`
  3. `+`,`-`
- In case of equal priority, execute from **left to right**.
- Use parentheses to overrule defaults or help readability

```
p = (n*0.0821*T) / ( V - n*b ) - ((n**2)*a / V**2)
```

Integer division

- Modify your program so that first 9 g of SO_2 are given
- Next the amount of gas is converted to moles (molecular weight is 64 g/mol)
- Use integer values
- Different from Fortran and Python2.X

Python source code

```
m_SO2 = 9 #g of SO2
n_SO2 = 9 / 64
print(" {:.2f} mol of SO2".format(n_SO2))
n_SO2 = 9 // 64
print(" {:.2i} mol of SO2".format(n_SO2))
```

Commenting a Python program

- Good programming habit!
- "#" denotes a comment, everything afterwards is ignored
- Use """ to document the purpose of a code or section

Python source code

```
""" Pressure of SO2 as VdW gas. DO NOT CHANGE a,b
    characteristic of SO2"""
a = 6.71 #SO2
b = 0.0564 #SO2
n = 0.5 #number of moles
V = 5 #volume in litre
T = 298.15 #temperature in K
p = n*0.0821*T / ( V - n*b ) - n**2*a / V**2
print (p)
```

Formatting the output:

Python source code

```
print ("T= {:6.2f} K -- P = {:6.3f} atm".format(T,p))
```

- {} represent slots in the string that will be replaced by arguments
- character ":" indicates that a format instruction follows (see next)
- old style formatting, still available in python3 but will be deprecated.

Formatting style

- Examples:

Formatting your output

```
print ("T= {:6.2f} K -- P = {:6.3f} atm".format(T,p))
```

g	compact notation
f	decimal (float) notation
6.3f	decimal notation, 3 decimals, field width is 6
.3f	decimal notation, 3 decimals, automatic width
e	scientific notation (e.g. 1.2e-4)
E	scientific notation (e.g. 1.2E-4)
5d	integer, field width is 5
s	string (text)

Standard mathematical functions

- Type in the ipython3 console:

```
x = 0.5
y = sqrt(x)
print(y)
```

- Exponential, Trigonometric, etc. functions require the *math* module
- Modules: increase python functionality but need to be explicitly imported

```
import math
x = 0.5
y = math.sqrt(x)
```

Extended example

Use of modules

```
import math
x = 0.5
y = math.sqrt(x)
# another possibility
from math import sqrt
y = sqrt(x)
# import several functions
from math import sort, sin
y = sin(sqrt(x))
# or
from math import * # import everything in math
y = sin(sqrt(x))
```

Let's work!!

Complete the first set of exercises

Strings

- Text in Python is contained in *strings* objects.
- Many programs need to manipulate text.

Assigning literal data

```
line = "Hello World!"
line2 = 'python is great!'
T_K = 298.15
value_str = str(T_K - 273.15)
print(value_str, type(value_str))
```

- Important to be fluent manipulating strings, input data is generally read as string!

Strings

- We can perform operations on strings.
- Strings also have specific **methods**.

```
line1 = 'Melting point of Fe: 1538 degrees '
line2 = ' at 1 atm.'
line = line1 + line2
line
len(line)
'Fe' in line
line.replace('Fe', 'Iron') #should better be assigned!
print(line)
line = line.replace('Fe', 'Iron')
print(line)
line.split() # generates a list, more on later on
line.split(':')
```

Operating with strings

- Symbolic calculation using strings:

```
codon1 = "atg"
codon2 = "atc"
vector = codon1 + codon2
seq = vector.upper() * 10
print(seq)
```

- String indexing: Specifying substrings

```
line1 = 'Melting point of Fe: 1538 degrees '
len(line1)
line1[0] #notice: first element has index 0
line1[len(line1)]
line1[len(line1) - 1]
line1[-1]
line1[17:]
line1[17:25]
```

Operating with strings

- Strings are inmutable:

```
codon = "atg"
codon[1] = "a"
```

- String methods:
  - Notice syntax: `varname.method(args)`
  - output should be assigned!

```
seq = 'cggggagtggggagttgagtcgcaagatgagcgagc'
seq.count('t') #number or tirosine
seq.upper() #capitalize (see .lower())
seq.split("a")
seq.find("g")
seq.replace("a","t")
```

Built-in Functions

- Built-in functions, hat a look at: Python docs
- Useful: `input()` function to **read from user keyboard input**

Reading input data

```
v = input('Insert a volume value: ')
type(v) # input is read as a string
v = float(v) # convert string to numeric value
coords = input('Coordinate values (use space)?')
x = float(coords.split()[0])
y = float(coords.split()[1])
print(x,y)
```

- More and more powerful functions? Python modules!

Boolean data

- `True` and `False`
- operators: ==, >, <, <=, >=, =!

Logical operations

```
3 == 4
3 != 4
"e" in "Hello"
"s" in "Hello"
```

- logical operators: `and`, `or`, `not`

Container data types

- Object that holds an arbitrary number of other objects.
    - lists
    - tuples
    - sets
    - dictionaries
- Later on, we will see that Scientific Python Modules provide additional container data types, specifically tailored for numerical computation.

Lists

- Values (or objects) can be grouped together in a `list`
- Lists can contain objects of different type, heterogeneous!
- Lists can grow or shrink dynamically
- Lists can have sublists as elements.

```
first = [ 2, 4, 6, 8, 10]
second = ['one', 'two', 'three']
third = list(range(3))
fourth = [ 2, 'one', False, first]
```

List indexing

- Elements are accessed the same way as in `strings`
- Access one element (indexing starts with 0!): `varname[i]`
- View/access a *slice* of the list: `varname[start:end:step]`
  Counterintuitive! `end` is excluded!

```
second[0] #first element
first[-1] #last element
second[1:3] # can you predict the result?
second[0:3] # can you predict the result?
second[:] # can you predict the result?
fourth[-1][1] #can you predict the result?
```

- Indexing in nested lists: `varname[i][j]`

```
fourth[-1][1] #can you predict the result?
```

More on Lists

- Lists are *mutable*, unlike strings

```
fourth[1] = 8
```

- Assignment between lists, creates **a reference**, not a copy

```
fifth = second
second[-1] = 15
fifth
```

- Copy the list!

```
fifth = second.copy()
second[-1] = 15
fifth
sixth = list(second)
sixth[0]= "wow!"
fifth, sixth
```

List operations

- Combining lists:

```
first + third
```

- Replicating lists:

```
3*third
```

- Logical operations: Is there an element in a list?

```
languages = ["Python", "C", "C++", "Java", "Perl"]
"C++" in languages
"HTML" in languages
```

Exercises:

- Does the last element in `languages` contain the letter "e"?
- How many "a"s does the second-to-last element contain?

List methods

List manipulation using methods

```
a = list(range(10))
a.sort(reverse=True)
print(a)
a.append(-5) #add an (-5) element at the end of the
    list
print(a)
a.pop() # produces -5
print(a) # is there a item missing?
a.pop(2) # produces 7, why?
a.index(5) #in which position can value (5) be found?
a.pop(a.index(5)) #can you predict the result?
print(a)
a.extend([12,13])
print(a)
```

- Use lists to represent arrays? better use Numpy

List methods

Extract a numerical value from a string

```
line = "Melting point of Fe: 1538 degrees"
words = line.split()
print(line)
tfus = words[-2]
print(type(tfus))
tfus = float(tfus)
print(type(tfus))
print(tfus+273.15)
```

Lists

- Lists are flexible but...
- Use lists to represent numerical *arrays* (vectors, matrices...)? better use `Numpy` arrays
- However, it is very convenient to know your way around with lists. It helps with Numpy arrays.

Tuples

- Together with `string` and `list` belong to the Sequence Type.
- A tuple consists of a number of values separated by commas.
- Similarly to strings and unlike lists, tuples are inmutable.

An example on tuples

```
t1 = (0.0821, 8.31416, 'Bolztman')
t2 = (6.023E23, 'Avogadro')
constants = t1 + t2
constants
constants[0] = 0.08214
```

Sets

- Python implementation of mathematical sets.
- unordered collection of objects.
- cannot have multiple occurrences of the same element.

```
s1 = set(['a', 'b', 'c', 'd'])
s2 = set(['c', 'd', 'e', 'f', 'g'])
s1.union(s2) # how many elements would you expect?
s1.intersection(s2)
s2.difference(s1)
s1.difference(s2) #non-commutative!
```

Dictionaries

- Dictionaries are an object called "mappings" or "associative arrays" in other languages.
- Based on keys, not ordered

A simple exercise with a dictionary: atomic masses

```
atomic_mass = {} # creates the dictionary (empty)
atomic_mass['H'] = 1
atomic_mass['C'] = 12
atomic_mass['N'] = 14
atomic_mass['O'] = 16
len(atomic_mass)
atomic_mass.keys() # notice, no ordering
atomic_mass['C']*2 + atomic_mass['O'] +
    atomic_mass['H']*6
del(atomic_mass['H'])
atomic_mass.clear()
```

Sequence control: conditional structure

- Different actions can be performed according to a **condition**. Example: Heaviside function

$$f(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$$

```
if x <= 0:
    fx = 0 # is executed if the condition is TRUE
else:
    fx = 1 # is executed if the condition is FALSE
print('function at {} is {}'.format(x,fx))
```

- The `else` part of an if-block can be skipped, if desired.

Sequence control: conditional sequence

- Comparison operators: `<`, `<=`, `>`, `>=`, `==`, `!=`, `is`

```
2 == 2
2 != 2
3 >= 2
-1 in [-1, 1]
ans = True
if ans: print('yes!')
```

- operators can be combined using boolean: `and`, `or`, `not`

```
if x >= 0 and x <= 1: print('Prob. within range')
```

Conditional sequence: multiple conditions

- Several mutually exclusive if tests:

```
""" This is the hat function """
   if x <= 0:
       hatx = 0
   elif 0 <= x < 1: #intuitive notation, just like
       math!
       hatx = x
   elif 1 <= x < 2:
       hatx = -x
   elif x >= 2:
       hatx = 0
   print('function at {} is {}'.format(x, hatx))
```

Conditional sequence: multiple conditions

- Several mutually exclusive if tests:

```
""" This is the hat function """
   if x <= 0:
       hatx = 0
   elif 0 <= x < 1: #intuitive notation, just like
       math!
       hatx = x
   elif 1 <= x < 2:
       hatx = -x
   elif x >= 2:
       hatx = 0
   print('function at {} is {}'.format(x, hatx))
```

- Have you noticed there is no `ENDIF`???

Sequence control: Python Blocks

- Block indentation is generally recommended for programming.
- Makes code clearer.
- This custom becomes **syntactic rule** in Python!
- Lines with the same indentation belong to the same block, once the block is done, back to original indentation.
- Python style rule: use *4 spaces* to indent
- Do not mix *tabs* and *spaces*!

Conditional sequence: multiple conditions

- Several mutually exclusive if tests:

```
""" This is the hat function """
   if x <= 0:
       hatx = 0
   elif 0 <= x < 1: #intuitive notation, just like
       math!
       hatx = x
   elif 1 <= x < 2:
       hatx = -x
   elif x >= 2:
       hatx = 0
   print('function at {} is {}'.format(x, hatx))
```

- Can you think of a shorter alternative?

Sequence control: conditional sequence

A more compact version

```
""" This is the hat function """
   if 0 <= x < 1:
       hatx = x
   elif 1 <= x < 2:
       hatx = -x
   else:       #covers both x<0 and x>=2!
       hatx = 0
   print('function at {} is {}'.format(x, hatx))
```

Sequence control: while loops

- Block is executed until the condition specified turns false.

```
while item < maxiter:
    x = f(x)
    iter += 1

print(x)
```

Sequence control: for loops to iterate over list elements

- For loops can nicely iterate over lists.

```
people = ['Susanna', 'Diego', 'Clara', 'Victor']
for name in people:
    print('Hi {}, have a nice day!'.format(name))
```

- We can also loop over **the positions** in a list!

```
people = ['Susanna', 'Diego', 'Clara', 'Victor']
npeople = len(people)
for i in range(npeople):
    print('Hi {}, have a nice day!'.format(people[i]))
```

- Loop over anything collective variable: Dictionaries, Sets...

Sequence control: for loops to iterate over list elements

- Print a list of temperature values converted from Celsius to Kelvin

Complete the code!

```
tCel = [-25.0, -15.0, 0, 15.0, 25.0]
for tC in tCel:
    tK = tC + 273.15
    print(tK)
```

Sequence control: for loops to iterate over list elements

- Build a (new) list with temperature values in Kelvin from values in Celsius

Complete the code!

```
tCel = [-25.0, -15.0, 0, 15.0, 25.0]
XXXXXX
for tC in XXXXXX:
    tK = XXXXXX + 273.15
    tKel.append(XXXXXX)
print(tKel)
```

Repetitive execution using for loops

- Execute a piece of code a given number of times:

a typical for loop

```
n = int(input("How many repetitions? "))
for ite in range(n):
    print("This iteration {}.".format(ite))
print("Done")
```

- Use a for loop to calculate a sum:

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \cdots = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

Sequence control: for loops

Summing using a for loop: complete the code

```
import XXXXXX
x = float(input("x? "))
nsum = input("How many terms in the sum? ")
XXXXXX
for n in XXXXXX:
    term = x**n
    series = XXXXXX
y = math.exp(x)
print("Series: {:.4f} // Exact: {:.4f}".format(series, y))
```

Using a for loop to build a list

Entering list elements one by one

```
nel = int(input("How many elements in your list? "))
my_list = [] #create an empy list!
for i in range(nel):
    element = input("Type your element: ")
    #Attention! Is it numeric? Convert it!
    my_list.append(element)
```

Using a for loop to build a list

Entering ALL LIST ELEMENTS AT ONCE

```
my_list = [] #create an empy list!
answer = input("Type all the elements in the list,
    separated by blank spaces: ")
my_list = answer.split()
```

- Notice that we did not ask for the number of elements!
- If the elements need to be numeric... use `float()`
- Exercise: Repeat the temperature conversion exercises, Celsius list is built from values given by the user (keyboard).
- Exercise (2): All values in the same line!

For loops to iterate over list elements

- Rewrite `p_vdw.py` and write a table of v,p values
    - User introduces Vmin, Vmax, N using keyboard

Sequence control: for loops

Revisiting the vdw program

```
a = 6.71 #SO2; b = 0.0564 #SO2
n = 0.5 #number of moles
T = 298.15 #temperature in K
str = input('Give Vmin and Vmax')
words = str.split()
vmin = float(words[0])
vmax = float(words[1])
nv = int(input('Number of values to calculate?'))
dv = (vmax - vmin)/(nv-1)
for i in range(nv):
    V = vmin + i*dv
    p = n*0.0821*T / ( V - n*b ) - n**2*a / V**2
    print ('{:5.2f} , {:5.2f}'.format(V,p))
```

List comprehensions

- A compact way of creating lists:

```
x_sq = [x**2 for x in range(10)]
```

Pythonic coding

```
x = [1, 2, -4, -5, 3, -1, 7]
x = [abs(y) for y in x]
```

- Multiple combinations:

```
[(x, y) for x in [1,2,3] for y in [3,1,4]]
[(x, y) for x in [1,2,3] for y in [3,1,4] if x!=y]
```

Useful commands in blocks

- `break`, `continue`, `pass`

A silly example

```
x = 1.0
i = 0
while True: #infinite loop!
    print(i,x)
    x -= 0.1
    i += 1 #update counter
    if x >= 0:
        pass  #do nothing
    else:
        break #exit loop
print("Done!")
```