Python Introduction

Dr. Rameshwar L. Kumawat
School of Chemistry & Biochemistry, Georgia Institute of Technology
GA-30332, USA

These notes were prepared during my postdoc days at Georgia Tech

Python is a computer programming language that has become omnipresent in scientific programming research. This initial lesson will run python interactively through a python interpreter.

Part 1. Very Basic but Important Stuff

1.1. Running Python Programs:

Python programs can be run from files similar to perl or shell scripts, e.g., by typing the following command line at the terminal:

$ python program.py
This file can contain just the python commands.

Program “Hello, world!”

Let’s get started with the basic example of a python program run as script”

```
(base) ram123@LAPTOP-TH1ME530:~$ python
Python 3.8.12 (default, Oct 12 2021, 13:49:34)
[GCC 7.5.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print ("Hello, world!")
Hello, world!
>>> exit()
```

Or if this program is saved to a file named: hello.py, then set execute permissions on this file and run it with:

```
(base) ram123@LAPTOP-TH1ME530:python hello.py
Hello, world!                # When run, this program prints
```

1.2. Printing

```
 >>> x = 4
>>> print ('hello:', x, x ** 2, x ** 3)
hello: 4 16 64
```

```
>>> exit()
```

Another Example: python basics.py
Input:

```
print (1 + 5)
pi = 3.1415926
print (pi)
message = ("Hello, world")
print (message)
```

output: $ python basics.py

```
6
3.1415926
Hello, world
```

1.3. Variable Types:

In the previous example, we have seen that “pi” and “message” are variables, but “pi” is a floating-point number, and “message” is a string.

Note that we did not declare any types in our example file “basics.py”. However, python has capably decided types for the variables.

Actually, in python “variables are *object references*. And, the reason we do not need to declare variable types is that a reference might point to a different type later.

Example Input: references.py

```
x = 12
y = "Hello"
print (x,y) #prints 12 Hello
print (x,y) #prints 12 12
```

Output:

```
(base) ram123@LAPTOP-TH1ME530:$ python references.py
12 Hello
12 Hello
```

Example Input: types.py

```
pi = 3.1415926
message = "Hello, world"
i = 4 + 4
print (type(pi))
print (type(message))
print (type(i))
```

Output:

```
(base) ram123@LAPTOP-TH1ME530:$ python types.py
<class 'float'>
<class 'str'>
<class 'int'>
```

Variables Names: It can contain letters, numbers, and underscores. It must begin with a letter. It cannot be one of the reserved python keywords, e.g.: and, as, assert, break, continue, class, def, del, else, elif,

exec, except, for, finally, from, global, import, if, is, in, lambda, or, not, pass, print, return, raise, try, with, while, yield
Names starting with one underscore (_V) are not imported from the module import * statement
Names starting and ending with 2 underscores are special, system-defined names (for example, _V_)
Names beginning with 2 underscores (but without trailing underscores) are local to a class (_V)
A single underscore (_) by itself denotes the result of the last expression

1.4. Operators: In general, operators are used to performing operations on variables and values.

+ (addition), - (subtraction), / (division), ** (exponentiation), % (modulus (remainder after division))

Example: operators.py
```
(base) ram123@LAPTOP-TH1ME530:$ cat operators.py
print (4*4)
print (4**5)
print (10%3)
print (3.0/4.0)
print (1//2) or print (int(1/2))
(base) ram123@LAPTOP-TH1ME530:$ python operators.py
16
1024
1
0.75
0
```

Note the difference between floating-point division and integer division in the last two lines.

+= but not ++

Python has incorporated operators like +=, but ++ or – don't work in python.

Type conversion
float(), int(), str(), and bool() convert to floating point, integer, string, and Boolean (True or False) types, respectively.

Example typeconv.py:	**Output**
print (1.0/2.0)	0.5
print (int(1/2))	0
print (float(1)/float(2))	0.5
print (int(3.1415926))	3
print (str(3.1415926))	3.1415926
print (bool(1))	True
print (bool(0))	False

Operators Acting on Strings

```
>>> "Ram!"*3
'Ram!Ram!Ram!'
>>> "Hello " + "Ram!"
'Hello Ram!'
```

```
>>> "Hello " + " , " + "Ram!"
'Hello  , Ram!'
```

1.5 Input from Keyboard:

```
Example input.py
i = raw_input("Enter a math expression: ")
print i
j = input("Enter the same expression: ")
print j
Output:
Enter a math expression: 3+2
3+2
Enter the same expression: 2+3
5
```

1.6. Comments: Anything after a # symbol is treated as a comment similar to Perl.

Part 2. Conditions: Python programming language supports the usual logical conditions from mathematics.

2.1. True and False Booleans,
2.2. Comparison and Logical Operators
2.3. If, elif, and else statements

2.1. Booleans: True and False

```
>>> type (True)
<class 'bool'>
>>> type (true)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'true' is not defined
>>> 2+2==5
False
```

Note that True and False are of type bool. *Capitalization* is required for Booleans.

Boolean Expressions:

- A Boolean expression can be evaluated as **True** or **False**.
- An expression evaluates to **False** if it is: the constant False, the object None, an empty sequence or collection, or a numerical item of value 0.
- Everything else is considered **True**.

Comparison Operators:

Operators	Description
== :	is equal to?

!= :	not equal to
> :	greater than
< :	less than
>= :	greater than or equal to
<= :	less than or equal to
is :	do two references refer to the same object?

Example:

- Can “chain” comparisons:
  >>> a = 12
  >>> 0 <= a <= 32
  True
  >>> 0 <= a <= 2
  False

Logical Operators:

- and, or, not
  Example:
  >>> 1+1==3 or 2+2==4
  True
  >>> 1+1==3 and 2+2==4
  False
  >>> not(1+1==3) and 2+2==4
  True

Note that we do NOT use &&, ||, !, as in C!

If Statements:

Example: ifelse.py

```
if (2+2==4):
   print ("2+2==4")
   print ("I always thought so!")
else:
   print ("My understanding of math must be faulty!")
```

Output:

```
(base) ram123@LAPTOP-TH1ME530:$ python ifelse.py
2+2==4
I always thought so!
```

elif statement:

Equivalent of “else if” in C!

Exaple: elif.py

```
x = 4
if (x==2):
   print ("two")
elif (x==3):
   print ("three")
```

```
else:
   print ("many")
```
Output:
(base) ram123@LAPTOP-TH1ME530:$ python elif.py
Many

Part 3. Functions: A function is a kind of block of code that only runs when it is called. We can pass data, known as parameters into a function. A function can return data as a result.

- **Defining functions**
- **Return values**
- **Local variables**
- **Built-in functions**
- **Functions of functions**
- **Passing lists, dictionaries, and keywords to functions**

Define them in the file above the point they are used
The body of the function should be indented consistently (4 spaces is typical in python)

Example: square.py
```
def square(n):
   return n*n
print ("The square of 3 is",)
print (square(3))
```
Output: $python square.py
The square of 3 is 9

The def statement

- The def statement is *executed* (that's why functions have to be defined before they are used)
- def creates an object and assigns a name to reference it; the function could be assigned another name, function names can be stored in a list, etc.
- Can put a def statement inside an if statement, etc!
- Arguments are optional.
- Multiple arguments are separated by commas.
- If there is no return statement, then "None" is returned.
- Return values can be simple types or tuples.
- Return values may be ignored by the caller.
- Functions are "typeless."
- Can call with arguments of any type, so long as the operations in the function can be applied to the arguments. This is considered as good thing in the python.

Function Variables are Local

- Variables declared in a function do not exist outside that function.

  Example: square_2.py
  ```
  def square(n):
          m = n*n
          return m
  print ("The square of 3 is",)
  ```

```
print (square(3))
print (m)
```

Output: python square_2.py
The square of 3 is
9
Traceback (most recent call last):
  File "square_2.py", line 6, in <module>
    print (m)
NameError: name 'm' is not defined

Scope:

Variables assigned within a function are local to that function call.
Variables assigned at the top of a module are global to that module; there's only "global" within a module.
Within a function, python will try to match a variable name to one assigned locally within the function; if that fails, it will try within enclosing function-defining (def) statements (if appropriate); if that fails, it will try to resolve the name in the global scope (but the variable must be declared global for the function to be able to change it). If none of these match Python will look through the list of build-in names.

Example: scope.py

```
a = 12 #global

def func(b):
    c = a+b
    return c
print (func(4)) #gives 4+12=16
print (c)       # not defined
```

Output: $python scope.py
16
Traceback (most recent call last):
  File "scope.py", line 7, in <module>
    print (c)   # not defined
NameError: name 'c' is not defined

Example: scope.py

```
a = 12 #global

def func(b):
    global c
    c = a+b
    return c

print (func(4)) #gives 4+12=16
print (c)       # now it's defined
```

Output: $python scope_1.py
16
16

By Reference / By Values:
Everything in python is a reference. However, note also that immutable objects are not changeable --- so changes to immutable objects within a function only change what object the name points to (add do not affect the caller, unless it's a global variable).

For immutable objects (e.g., integers, strings, tuples), python acts like C's pass by vales.
For mutable objects (e.g., lists), python acts like C's pass-by point; in-place changes to mutable objects can affect the caller.

Example: passbyref.py

```
def f1(x,y):
    x = x*1
    y = y*2
    print (x,y) # 0 [1, 2, 1, 2]

def f2(x,y):
    x = x*1
    y[0] = y[0]*2
    print (x,y) #0 [2, 2]
a = 0
b = [1,2]
f1(a,b)
print(a,b) #0 [1, 2]
f2(a,b)
print (a,b) # 0 [2, 2]
```

Output: $ python passbyref.py

```
0 [1, 2, 1, 2]
0 [1, 2]
0 [2, 2]
0 [2, 2]
```

Multiple Return Values:
Can return multiple values by packaging them into a tuple
Example: multiretvalue.py

```
def onetwothreefourfive(x):
    return x*1, x*2, x*3, x*4, x*5
print (onetwothreefourfive(5))
```

Output: $ python multiretvalue.py

```
(5, 10, 15, 20, 25)
```

Built-in Functions:
Several useful build-in functions.
Example: math.py

```
print (pow(3,4))
print (abs(-15))
print (max(1,-6,4,0))
```

Output: $ python math.py

```
81
15
```

4

Functions of Functions:

Example: funcfunc.py
```
def iseven(x,f):
    if (f(x) == f(-x)):
        return True
    else:
        return False

def square(n):
    return(n*n)
def cube(n):
    return(n*n*n)
print (iseven(2,square))
print (iseven(2,cube))
```
Output: $ python funcfunc.py
True
False

Default Arguments: Python programming has a different way of representing syntax and default values for function arguments. Default values indicate that the function argument will take that value if no argument value is passed during the function cell. The default value is assigned by using the assignment(=) operator of the from keywordname=value.
Like C or Java program, we can define a function to supply a default value for an argument if one isn't specified.

Functions Without Return Values: All functions in the Python program return something. If a return statement is not given then by default python program returns None.
Beware of assigning a variable to the result of a function which returns None. For example the list append function changes the list but does not return a value:

Example:
```
>>> a = [0,1,2]
>>> b = a.append(3)
>>> print (b)
None
```

Part 4. Iteration: Repeated execution of a set of statements is called iteration. Iteration is so common and Python provides several language features to make it easier.

- **while loops**
- **for loops**
- **range function**
- **Flow control within loops: pass, continue, break, and the "loop else"**

while loops:
(i) Example: while.py

```
i = 1
while i < 5:
   print (i)
   i += 1
```
Output: $ python while.py
1
2
3
4

(ii) Example:
for loops:
Example: for.py
```
for i in range(5):
   print (i),
```
Output: $ python for.py
0
1
2
3
4
range(n) returns a list of integers from 0 to n-1.
range(0,10,2) returns a list 0, 2, 4, 6, 8

Flow control within loops:
General structure of a loop:
```
while <statement> (or for <item> in <object>):
        <statements within loop>
        if <test1>: break# exit loop now
        if <test2>: continue        # go to top of loop now
        if <test3>: pass  # does nothing!
else:
        <other statements>        # if exited loop without hitting a break
```

Using the "loop else"
An else statement after a loop is useful for taking care of a case where an item is not found in a list.
Example: search_item.py
```
for i in range(4):
   if i == 5:
      print ("I found 5!")
      break
   else:
      print ("Do not care about!")
else:
    print ("I searched but never found 5!")
```
Output: $ python search_item.py
Do not care about!
Do not care about!
Do not care about!
Do not care about!

I searched but never found 5!

for … in loops:
Used with collection data types which can be iterated through (“iterables”):
Example: forin.py

```
for name in ["Ram", "Ryan", "Krishan"]:
    if name[0] == "R":
        print (name, "starts with an R")
    else:
        print (name, "doesn't start with R")
```

Output: $ python forin.py

```
Ram starts with an R
Ryan starts with an R
Krishan doesn't start with R
```

Parallel Traversals:
If we want to go through 2 lists in parallel, one can use zip:
Example: zip.py

```
A = [1, 2, 3, 4, 5, 6]
B = [4, 5, 6, 7, 8, 9]
for (a,b) in zip(A,B):
    print (a, "*", b, "=", a*b)
```

Output: $ python zip.py

```
1 * 4 = 4
2 * 5 = 10
3 * 6 = 18
4 * 7 = 28
5 * 8 = 40
6 * 9 = 54
```

Part 5. Strings: Strings in python are surrounded by either single quotation marks or double quotation marks.

- **String basics**
- **Slices**
- **Escape sequences**
- **Block quotes**
- **Formatting**
- **String methods**

String basics:
Strings can be delimited by single or double quotes.
Python uses Unicode, so strings are not limited to ASCII characters.
An empty string is denoted by having nothing between string delimiters (e.g., '').
Can access elements of strings with [], with indexing starting from zero:

```
>>> "snakes"[3]
'k'
>>> "snakes"[0]
```

's'

Note that it can not go other way --- can not set “snakes”[3] = ‘p’ to change a string; strings are immutable.
a[-1] gets the *last element* of string a (negative indices work through the string backwards from the end).
Example:
>>> "snakes"[-1]
's'
>>> "snakes"[-3]
'k'

More string basics:
Type conversion:
>>> int("12")
12
>>> str(12.4)
'12.4'
Compare strings with the is-equal operator, == (like in C and C++):
>>> a = "hello"
>>> b = "hello"
>>> a == b
True
>>> location = "Jaipur " + " Rajasthan"
>>> location
'Jaipur Rajasthan'

Escape sequences

Escape	Meaning
\\	\
\'	'
\”	“
\n	Newline
\t	tab
\N{id}	Unicode dbase id
\uhhhh	Unicode 16-bit hex
\Uhhhh…	Unicode 32-bit hex
\x	Hex digits value hh
\0	Null byte (unlike C, does not end string)

Block quotes:
Multi-line strings use triple-quotes:
>>> lincoln = """I came to the USA two months ago,
... a new nation, for postdoctoral studies."""

String formatting:
Formatting syntax:
Format % object-to-format

>>> greeting = "Hello"
>>> "%s. Welcome to Python." % greeting

'Hello. Welcome to Python.'

Note that formatting creates a new string because strings are immutable.
The advanced printf-style formatting from C works. It can format multiple variables into one string by collecting them in a tuple (comma-separated list delimited by parentheses) after the % character:
>>> "The grade for %s is %5.1f" % ("Ram", 86.052)
'The grade for Ram is 86.1'
String formats can refer to dictionary key:
>>> "%(name)s got a %(grade)d" % {"name":"Ram","grade":92.5}
'Ram got a 92'

Stepping through a string:
A string is treated as a collection of characters and so it has some properties in common with other collections like lists and tuples.
Example:
>>> for d in "snakes": print (d),
...
s
(None,)
n
(None,)
a
(None,)
k
(None,)
e
(None,)
s
(None,)
>>> 'a' in "snakes"
True
>>> 'k' in "snakes"
True

Slices:
Slice: get a substring from position i to j-1:
>>> "snakes"[1:3]
'na'
>>> "snakes"[0:2]
'sn'
Both arguments of a slice are optional; a[:] will just provide a copy of string a.

String methods:
Strings are classes with many build-in methods. Those methods that create new strings need to be assigned (since strings are immutable, they cannot be changed in-place).
S.capitalize(), S.center(width), S.conut(substring [, start-idx [, end-idx]]),
S.find(substring [, start [, end]]))
S.isalpha(), S.isdigit(), S.islower(), S.isspace(), S.isupper()
S.join(sequence)
etc

Replace method:
Does not really replace (strings are immutable) but makes a new string with the replacement performed:
Example:

```
>>> a = "ABCDeFGHIJkLMNOPQRsTUVWXYZ"
>>> b = a.replace('s', 'S')
>>> b
'ABCDeFGHIJkLMNOPQRSTUVWXYZ'
>>> a
'ABCDeFGHIJkLMNOPQRsTUVWXYZ'
```

More methods examples:
methods.py:

```
a = "I found it boring and I left"
loc = a.find("boring")
a = a[:loc] + "fun"
print (a)

b = ' and '.join(["tractors", "cars", "trucks"])
print (b)

c = b.split()
print (c)

d = b.split(" and ")
print (d)
```

Output: $ python methods.py

```
I found it fun
tractors and cars and trucks
['tractors', 'and', 'cars', 'and', 'trucks']
['tractors', 'cars', 'trucks']
```

Regular Expressions:
Regular expressions are a way to do pattern-matching. The basic concept is the same as Java or perl. Most of the syntax of the actual regular expression is the same in Java or perl.

Regular Expression Syntax
Common regular expression syntax:

Parameter	Description
.	Matches any char but newline (by default
^	Matches the start of a string
$	Matches the end of a string
*	Any number of what comes before this
+	One or more of what comes before this
|	Or
\w	Any alphanumeric character
\d	Any digit
\s	Any whitespace character
\W	Matches NON-alphanumeric

\D	NON digits
[aeiou]	Matches any of a, e, i, o, u
junk	Matches the string 'junk'

Example 1: regex.py

```
import re
txt = "The rain in USA"
x = re.search("^The.*USA$", txt)

if x:
   print("Yes!, We have a match!")
else:
   print("We don't have a match!")
```

Output: $ python regex.py
Yes!, We have a match!

Example 2: regex_1.py

```
import re
pattern = '^a...s$'
test_string = 'abyss'
result = re.match(pattern, test_string)
if result:
   print("Search successful.")
else:
   print("Search unsuccessful.")
```

Output: $ python regex_1.py
Search successful.

Note that here we have used **re.match()** function to search **pattern** within the **test_string**. This method returns a match object if the search is successful. If not, it returns **None**.

The *Match* Function: This function attempts to match Re pattern to string with optinal flags. Here in the syntax for this function: re.match(pattern, string, flags=0)
Pattern: this is the regular expression to be matched.
String: this is the string which would be searched to match the pattern at the beginning of string.
Flags: we can specify different flags using bitwise OR (|). These are modifiers which are listed below.
The re.match function returns a **match** object on success, **None** on failure. We usegroup(num) or groups() function of match object to get matched expression.
group(num=0): this method returns entire match (or specific subgroup num)
groups(): this method returns all matching subgroups in a tuple (empty if there were not any)
Example: regex_2.py

```
import re
line = "Humans are smarter than animals"
matchObj = re.match( r'(.*) are (.*?) .*', line, re.M|re.I)
if matchObj:
   print ("matchObj.group() : ", matchObj.group())
   print ("matchObj.group(1) : ", matchObj.group(1))
   print ("matchObj.group(2) : ", matchObj.group(2))
```

```
else:
   print ("No match!!")
```

```
Output: $ python regex_2.py
matchObj.group() :  Humans are smarter than animals
matchObj.group(1) :  Humans
matchObj.group(2) :  smarter
```

to be continued…in next lecture notes