Astronomical data analysis using Python

Lecture 3

Our first program - introduced in the previous lecture

```
In [126]: a = 3
b = 5
c = a+b
d = a-b
q, r = a/b, a%b # Yes, this is allowed!

# Now, let's print!
print ("Hello World!") # just for fun
print ("Sum, Difference = ", c, d)
print ("Quotient and Remainder = ", q, r)
```

```
Hello World!
Sum, Difference =  8 -2
Quotient and Remainder =  0.6 3
```

Our First Program - Rewritten!

Let us introduce the following modifications to the program.

- We use floats instead of ints.
- We accept the numbers from the user instead of "hard coding" them.

In [65]:
```
# Modified first program.
a = input("Please enter number 1: ")
b = input("Please enter number 2: ")

c, d = a+b, a-b
q, i = a/b, a//b

print (c,d,q,i)
```

```
Please enter number 1: 3
Please enter number 2: 5

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-65-8d9a5e1f3c2a> in <module>
      3 b = input("Please enter number 2: ")
      4 
----> 5 c, d = a+b, a-b
      6 q, i = a/b, a//b
      7 

TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

What happened?

- Anything input through the keyboard using input() is ... a "string".
- Strings support addition (concatenation) but nothing else.

So what should we do?

- "3.0" is a string. 3.0 is a float!
- To convert "3.0" into a float, we use a simple function - float("3.0")

So, let's rewrite our program!

```
In [127]: a = float( input("Enter Number 1: ") ) # We are nesting functions here.
          b = float( input("Enter Number 2: ") )

          c,d = a+b, a-b
          q,i = a/b, a//b # a//b is the floor division operator

          print ("Addition = %f, Difference = %f " % (c,d))
          print ("Quotient = %f, Floor division quotient  = %f" % (q,i))
```

```
Enter Number 1: 3
Enter Number 2: 5
Addition = 8.000000, Difference = -2.000000
Quotient = 0.600000, Floor division quotient  = 0.000000
```

The output looks ugly. Wish I could control the number of decimal places...

```
In [128]: a = float( input("Enter Number 1: ") )
          b = float( input("Enter Number 2: ") )

          c,d = a+b, a-b
          q,i = a/b, a//b

          print("Addition = %.2f, Difference = %.2f " % (c,d))
          print("Quotient = %.2f, Floor division quotient = %.2f" % (q,i))
```

```
Enter Number 1: 3
Enter Number 2: 5
Addition = 8.00, Difference = -2.00
Quotient = 0.60, Floor division quotient = 0.00
```

Ah! now, that's much better.

String Formatting

We have seen a powerful of constructing strings in the previous example.

```
In [68]: print ("Addition = %.2f, Difference = %.2f " % (c,d))
```

```
Addition = 8.00, Difference = -2.00
```

C / FORTRAN users amongst you will immediately understand this method of string construction.

Python supports this and its own way of string formatting.

```
In [69]: gal_name = "NGC 7709"; int_bmagnitude = 13.6
```

```
In [70]: statement1 = "The galaxy %s has an integrated \
         B-band magnitude of %.2f" % (gal_name, int_bmagnitude)
```

```
In [71]: statement2 = "The galaxy {0:s} has an integrated \
         B-band magnitude of {1:.2f}".format(gal_name, int_bmagnitude)
```

```
In [72]: statement3 = "The galaxy {name:s} has an integrated \
         B-band magnitude of {mag:.2f}".format(name=gal_name, mag=int_bmagnitude)
```

All the above statements are equivalent!

```
In [73]: print (statement1,"\n", statement2, "\n", statement3, "\n")
```

```
The galaxy NGC 7709 has an integrated B-band magnitude of 13.60
 The galaxy NGC 7709 has an integrated B-band magnitude of 13.60
 The galaxy NGC 7709 has an integrated B-band magnitude of 13.60
```

You can choose whichever method you like!

As a former C/C++ user, I tend to use the first method.

But ... second and third methods are more "Pythonic". If you don't have previous experience with the C type formatting use one of the more Pythonic ways.

Raw Strings

- We have seen the three methods of string declaration.
- We have also seen string formatting.
- String formatting taught us that symbols like { or % have special meanings in Python.

In [74]:
```
# There is a special way of declaring strings where
# we can ask Python to ignore all these symbols.
raw_string = r"Here is a percentage sign % and a brace }"
print (raw_string)
```

```
Here is a percentage sign % and a brace }
```

Usefulness of Raw Strings - Example

- Typically, when we make plots and set labels, we would like to invoke a LaTeX parser.
- This would involve a lot \ $ and {}.

In such cases, it's best to use raw strings.

```
plt.xlabel(r" \log \rm{F}_v")
```

Other examples with special characters include writing Windows file paths, XML code, HTML code, etc.

Conditionals

```
In [75]:
num = int(input("Enter number: ") )
if num %2 == 0:
    print ("%d is even!" % num)
else:
    print ("%d is odd!" % num)
```

```
Enter number: 3
3 is odd!
```

You will use conditionals a lot in your data analysis.

```
In [1]: model_choice = int(input( "Enter choice [1 or 2]: ") )
        spectrum = 3 # In a realistic case, this will be some complicated object.

        if model_choice == 1:
            #model1(spectrum)
            print ("Model 1 fitted.")
        elif model_choice == 2:
            #model2(spectrum)
            print ("Model 2 fitted.")
        else:
            print ("Invalid model entered.")
```

```
Enter choice [1 or 2]: 2
Model 2 fitted.
```

What do you notice apart from the syntax in the above example?

Indentation - A Vital Part of the Pythonic Way

Be it the if-block illustrated above or the loops or the functions (to be introduced soon), indentation is at the heart of the Python's way of delimiting blocks!

Function definitions, loops, if-blocks - nothing has your typical boundaries like { } as in C/C++/Java.

The "level of the indentation" in the only way to define the scope of a "block".

In support of indentation

Look at the following C-like code.

```
if (x>0)
    if (y>0)
        print "Woohoo!"
else
        print "Booboo!"
```

Which "if" does the "else" belong to?

In C and C-like languages, the braces {}s do the marking, the indentation is purely optional. In Python, indentation levels determine scopes. In Python the "the else" belongs to "if (x>0)".

Python forces you to write clean code! (Obfuscation lovers, go take a jump!)

Use either spaces or tabs (don't ever ever mix them) and use them consistently. I strongly recommend using 4 spaces for each level of indentation.

Wrapping up if-elif-else

The general syntax:

```
if <condition>:
    do this
    and this
elif <condition>:
    this
    and this
...
else:
    do this
    and this
```

Conditions are anything that return True or False.

- == (equal to)
- !=
- >
- >=
- <
- <=

You can combine conditionals using "logical operators"

- and
- or
- not

The Boolean Data Type

In [5]:
```
a = True
b = False

if a:
    print ("This comes on screen.")

if b:
    print ("This won't come on screen.")

if a or b:
    print ("This also comes on screen.")

if a and b:
    print ("This won't come on screen either.")
```

```
This comes on screen.
This also comes on screen.
```

In [78]:
```
type(a) # To check type of object.
```

Out[78]:
```
bool
```

Almost all other types have a Boolean Equivalent

```
In [6]:  a = 1
         b = 0

         if a:
             print ("Hello!")
         if b:
             print ("Oh No!")

         type(a)

         Hello!

Out[6]:  int
```

```
In [80]: s1 = ""; s2 = "Hello" # s1 is an empty string

         if s1:
             print ("Won't be printed.")
         if s2:
             print ("Will be printed.")
```

```
Will be printed.
```

This is bad Python style because remember that explicit is better than implicit. Use an expression that evaluates to a boolean instead. Keep your programs readable.

Conditional Expression

Consider...

```
In [81]: if 5 > 6:
             x = 2
         else:
             x = 3
```

```
In [82]: y = 2 if 5 > 6 else 3 # if else block in one line is allowed
```

```
In [83]: print (x,y)
```

```
3 3
```

A Second Plunge into the Data Types

The two other data types we need to know:

- Lists
- Dictionaries

Data Types I will not cover (formally):

- Tuples (immutable lists!)
- Sets (key-less dictionaries!)
- Complex Numbers
- Fractions
- Decimals

Lists

```
In [84]: a = [1,2,3,4] # simple ordered collection
```

```
In [85]: b = ["Hello", 45, 7.64, True] # can be heterogeneous
```

```
In [86]: a[0], a[-1], a[1:3] # All "sequence" operations supported.

Out[86]: (1, 4, [2, 3])
```

```
In [87]: b[0][1] # 2nd member of the 1st member

Out[87]: 'e'
```

```
In [88]: a = [ [1,2,3] , [4,5,6] , [7,8,9] ] # list of lists allowed.

In [89]: a[2][1] # Accessing elements in nested structures.

Out[89]: 8

In [90]: [1,3,4] + [5,6,7] # Support concatenation

Out[90]: [1, 3, 4, 5, 6, 7]

In [91]: [1,6,8] * 3 # Repetition (like strings)

Out[91]: [1, 6, 8, 1, 6, 8, 1, 6, 8]
```

Lists are Mutable! (Strings are not!)

```
In [9]: a = [1,4,5,7]
```

```
In [10]: print (a)
```

```
[1, 4, 5, 7]
```

```
In [11]: a[2] = 777 # set third element to 777
```

```
In [12]: print (a)
```

```
[1, 4, 777, 7]
```

List Methods

```
In [13]: a = [1,3,5]
         print (a)

[1, 3, 5]
```

```
In [97]: a.append(7) # adds an element to the end
         print (a) # the list has changed (unlike string methods!)

[1, 3, 5, 7]
```

```
In [98]: a.extend([9,11,13]) # concatenates a list at the end
         print (a)

[1, 3, 5, 7, 9, 11, 13]
```

```
In [99]: a.pop() # Removes one element at the end.
         print (a)

[1, 3, 5, 7, 9, 11]
```

```
In [100]: a.pop(2) # Removes 3rd element.
          print (a)

[1, 3, 7, 9, 11]
```

Don’t Forget!!!

In [101]:
```
print (dir(a)) # list of methods for a list "a"
```

```
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__',
'__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__geta
ttribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__i
nit__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul
__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__revers
ed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__s
ubclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'inser
t', 'pop', 'remove', 'reverse', 'sort']
```

In [102]:
```
help(a.sort)
```

```
Help on built-in function sort:

sort(*, key=None, reverse=False) method of builtins.list instance
    Sort the list in ascending order and return None.

    The sort is in-place (i.e. the list itself is modified) and stable (i.e. t
he
    order of two equal elements is maintained).

    If a key function is given, apply it once to each list item and sort them,
    ascending or descending, according to their function values.

    The reverse flag can be set to sort in descending order.
```

Implications of Mutability

```
In [103]: l = [1,2,3,4]
          m = l

          l.append(5)
          print (l)
          print (m)
```

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
```

l and m point to the same object. When the object mutates, whether you refer to it using l or m, you get the same mutated object.

How do I make a copy then?

In [104]:
```
l = [1,2,3,4]
m = l[:]

l.append(5)
print (l)
print (m)
```

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4]
```

Python has a module called "copy" available for making copies. Refer to the standard library documentation for details.

Dictionaries

- Imagine a list as a collection of objects obj0, obj1, obj2 ...
- First object has a location 0, second 1 ...
- Now, imagine renaming location 0 as "something", location 1 as "somethingelse" ...
- Earlier, you accessed objects at numbered locations a[0].
- Now, you access objects by specifying location labels a["something"]

Let's see this at work.

```
In [105]: d1 = { "a" : 3, "b" : 5}
          print (d1["a"])
          print (d1["b"])
```

```
3
5
```

"a", "b" are called keys and 3,5 are called values. So formally, a dictionary is a collection of key-value pairs.

```
In [106]: d1["c"] = 7 # Since "c" does not exist, a new key-value pair is made.
          d1["a"] = 1 # Since "a" exists already, its value is modified.
          print (d1) # the ordering of key-value pairs in the dictionary is not guarantee
          d.
```

```
{'a': 1, 'b': 5, 'c': 7}
```

Dictionary Methods

```
In [107]: keys = d1.keys() # Returns a list of all keys which is stored in "keys".
          print (keys)

          dict_keys(['a', 'b', 'c'])
```

```
In [108]: values = d1.values() # Returns a list of values.
          print (values)

          dict_values([1, 5, 7])
```

```
In [109]: d1.items() # List of Tuples of key-value pairs.

Out[109]: dict_items([('a', 1), ('b', 5), ('c', 7)])
```

Defining Dictionaries - ways to do this

In [110]:
```
d1 = {"a":3, "b":5, "c":7} # we've seen this.
```

In [111]:
```
keys =  ["a", "b", "c"]
values = [3,5,7]
d2 = dict( zip(keys,values) ) # creates dictionary similar to d2
```

In [112]:
```
d3 = dict( a=3, b=5, c=7) # again, same as d1,d2
```

In [14]:
```
d4 = dict( [ ("a",3), ("b",5), ("c",7) ] ) # same as d1,d2,d3b
```

Dictionaries in data analysis

```
z['M31']

rmag['M1']

lumdist['3C 273']
```

The while loop

```
In [114]: x = 0
          while x<5:
              print (x)
              x += 1
```

```
0
1
2
3
4
```

In [115]:

```python
x = 1
while True: # Without the break statement this loop is infinite
    print ("x = %d" % x)
    choice = input("Do you want to continue? ")
    if choice != "y":
        break # This statement breaks the loop.
    else:
        x += 1
```

```
x = 1
Do you want to continue? y
x = 2
Do you want to continue? y
x = 3
Do you want to continue? n
```

The ”for” loop - Pay Attention!

```
In [116]: x = [5,6,7,8,9,0] # a simple list
          for i in x:
              print (i)
```

```
5
6
7
8
9
0
```

In " for i in x", x can be many different things, any iterable object is allowed.

```
In [117]: s = "Hello!"

          for c in s:
              print (c)
```

```
H
e
l
l
o
!
```

No No No! I want my good old for-loop back which generates numbers x to y in steps of z!!!

In [118]:
```python
# OKAY!!! Let's try something here.

for i in range(2,15,3):
    print (i)
```

```
2
5
8
11
14
```

Let us see some wicked for-loops.

In [17]:
```
a = [1,2,3,4,5]
b = "Hello"
c = zip(a,b)
print(type(c))

for i,j in c:
    print (i, j)
```

```
<class 'zip'>
1 H
2 e
3 l
4 l
5 o
```

In [133]:
```python
a = "Hello!"

for i, c in enumerate(a):
    print ("Character no. %d is %s" % (i+1, c)) # i+1 is used because Python is 0-indexed.

print()
help(enumerate)
```

```
Character no. 1 is H
Character no. 2 is e
Character no. 3 is l
Character no. 4 is l
Character no. 5 is o
Character no. 6 is !

Help on class enumerate in module builtins:

class enumerate(object)
 |  enumerate(iterable, start=0)
 |
 |  Return an enumerate object.
 |
 |    iterable
 |      an object supporting iteration
 |
 |  The enumerate object yields pairs containing a count (from start, which
 |  defaults to zero) and a value yielded by the iterable argument.
 |
 |  enumerate is useful for obtaining an indexed list:
 |      (0, seq[0]), (1, seq[1]), (2, seq[2]), ...
 |
 |  Methods defined here:
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
    |
    |  __iter__(self, /)
    |      Implement iter(self).
    |
    |  __next__(self, /)
    |      Implement next(self).
    |
    |  __reduce__(...)
    |      Return state information for pickling.
    |
    |  ----------------------------------------------------------------------
    |  Class methods defined here:
    |
    |  __class_getitem__(...) from builtins.type
    |      See PEP 585
    |
    |  ----------------------------------------------------------------------
    |  Static methods defined here:
    |
    |  __new__(*args, **kwargs) from builtins.type
    |      Create and return a new object.  See help(type) for accurate signatur
   e.
```

You can break and continue for-loops too!

In [18]:
```
for i in range(10000):
    if i%2 == 0: # Even
        print (i,"is Even")
        continue
    print (i, "is Odd")

    if i == 7: # What if I had said "i==8 or i==10" ??????
        break
```

```
0 is Even
1 is Odd
2 is Even
3 is Odd
4 is Even
5 is Odd
6 is Even
7 is Odd
```

Traversing Dictionaries using for-loops

In [136]:

```
d = dict( a = 1, b = 2, c = 3, d = 4)

for key,value in d.items():
    print (key, "-->", value)
```

```
a --> 1
b --> 2
c --> 3
d --> 4
```

In [123]:

```
for key in d.keys():
    print (key, "-->", d[key])
```

```
a --> 1
b --> 2
c --> 3
d --> 4
```

Style Guide for Python Code

The PEP 8 provides excellent guidance on what to do and what to avoid while writing Python code. I strongly urge you to study PEP 8 carefully and implement its recommendations strictly.

https://www.python.org/dev/peps/pep-0008/ (https://www.python.org/dev/peps/pep-0008/)

The PEP 8 is very terse. For a more expanded explanation see:

https://realpython.com/python-pep8/ (https://realpython.com/python-pep8/)

Python is fully Unicode UTF-8 standard compliant

```
In [3]: print("What is your name?")
        print("আপনার নাম কি?")
        print("உங்கள் பெயர் என்ன?")
        print("तुझं नाव काय आहे?")
        print("तुम्हारा नाम क्या हे?")
```

```
What is your name?
আপনার নাম কি?
உங்கள் பெயர் என்ன?
तुझं नाव काय आहे?
तुम्हारा नाम क्या हे?
```

Google provides the Cloud Translation API client library for Python

https://cloud.google.com/translate/docs/reference/libraries/v2/python
(https://cloud.google.com/translate/docs/reference/libraries/v2/python)