Introduction to Python

Lecture 1

Arul Lakshminarayan, 22/9/17

What is Python?

- A reptile
- Part of the title of a well-known BBC comedy skit: Monty Python
- A programming language

Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. It was created by Guido van Rossum during 1985- 1990. Like Perl, Python source code is also available under the GNU General Public License (GPL).

- I am using IDLE and Python 3.6

```
Python 3.6.0 (v3.6.0:41df79263a11, Dec 22 2016, 17:23:13)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> WARNING: The version of Tcl/Tk (8.5.9) in use may be unstable.
Visit http://www.python.org/download/mac/tcltk/ for current information.
```

- Available freely for Linux, Windows, Mac OS.
- IDLE allows interactive programming.
- IDLE also has an inbuilt editor.
- Files are to saved as *.py

- start with prompt >>>
- If using Anaconda Navigator, use “qtconsole” and input reads like a Mathematica notebook: “In[1]:” etc..

```
Jupyter QtConsole 4.3.0
Python 3.6.1 |Anaconda 4.4.0 (x86_64)| (default, May 11 2017, 13:04:09)
Type "copyright", "credits" or "license" for more information.

IPython 5.3.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]:
```

From the Python tutorial @python.org

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

The Python interpreter and the extensive standard library are freely available in source or binary form for all major platforms from the Python Web site, https://www.python.org/, and may be freely distributed. The same site also contains distributions of and pointers to many free third party Python modules, programs and tools, and additional documentation.

For a description of standard objects and modules, see The Python Standard Library. The Python Language Reference gives a more formal definition of the language. To write extensions in C or C++, read Extending and Embedding the Python Interpreter and Python/C API Reference Manual. There are also several books covering Python in depth.

Python enables programs to be written compactly and readably. Programs written in Python are typically much shorter than equivalent C, C++, or Java programs, for several reasons:

- the high-level data types allow you express complex operations in a single statement;
- statement grouping is done by indentation instead of beginning and ending brackets;
- no variable or argument declarations are necessary.

By the way, the language is named after the BBC show “Monty Python’s Flying Circus” and has nothing to do with reptiles. Making references to Monty Python skits in documentation is not only allowed, it is encouraged!

- **int (signed integers)** – They are often called just integers or **ints**. They are positive or negative whole numbers with no decimal point. Integers in Python 3 are of unlimited size. Python 2 has two integer types - int and long. There is no '**long integer**' in Python 3 anymore.

- **float (floating point real values)** – Also called floats, they represent real numbers and are [illegible] a decimal point dividing the integer and the fractional parts. Floats may also be in scientif[illegible] with E or e indicating the power of 10 (2.5e2 = 2.5 x 10^2 = 250).

- **complex (complex numbers)** – are of the form a + bJ, where a and b are floats and J (or j[illegible] the square root of -1 (which is an imaginary number). The real part of the number is a, and[illegible] imaginary part is b. Complex numbers are not used much in Python programming.

```
>>> 2
2
>>> 2.0
2.0
>>> 2**30
1073741824
>>> 2.0**30
1073741824.0
>>> type(2)
<class 'int'>
>>> type(2.0)
<class 'float'>
>>> 2+4j
(2+4j)
>>> type(2+4j)
<class 'complex'>
>>> 2+4 j
SyntaxError: invalid syntax
>>> 2+j 4
SyntaxError: invalid syntax
>>>
```

Using the interpreter as a calculator

```
>>> 1+2
3
>>> 2*3
6
>>> 2.0*3.4
6.8
>>> 3.0/4.0
0.75
>>> 3/4
0.75
>>> 2*(3.4-4.5)/6
-0.3666666666666667
>>> 2   /  4
0.5
>>> 2**7
128
>>> 2**190
1569275433846670190958947355801916604025588861116008628224
>>>
```

```
>>> c
(4+3j)
>>> c.real
4.0
>>> c.imag
3.0
>>> c.conjugate()
(4-3j)
>>> c.conjugate
<built-in method conjugate of complex object at 0x1023bd8d0>
>>>
```

c is an object and c.real is an enquiry of the value of a component of the object, it does not create new objects.
c.conjugate is a function that either alters the object or creates a new one.

More on complex numbers

```
>>> # Alternative way to represent complex numbers ...
>>> z= 4+ 5j
>>> z1=complex(4,5)
>>> z==z1
True
>>> z is z1
False
>>> id(z)
4332443760
>>> id(z1)
4381605104
>>> z*z.conjugate()
(41+0j)
```

```
>>> z*z.conjugate().real
(16+20j)
>>> (z*z.conjugate()).real
41.0
>>>
```

```
>>> z*z.conjugate
Traceback (most recent call last):
  File "<pyshell#61>", line 1, in <module>
    z*z.conjugate
TypeError: unsupported operand type(s) for *: 'complex' and 'builtin_function_or_method'
>>>
```

Comments, Remainders (Modulo), Floor divisions

```
>>> #This is a comment,
>>> #Division returns floating point numbers (beware earlier versions esp. 2.* do not do so,
>>> 50/3
16.666666666666668
>>> ##Here are additional operations: if n= a* m +b, then a=n // m and b = n%m, that is "a" is the floor division and "b" the remainder (also n modulo m)
>>> 50 // 3
16
>>> 50 % 3
2
>>>
```

Exercises:
50/3/2
30%4%3
40 X (34.5 -3)^2/4+3

Comments, Remainders (Modulo), Floor divisions

```
>>> #This is a comment,
>>> #Division returns floating point numbers (beware earlier versions esp. 2.* do not do so,
>>> 50/3
16.666666666666668
>>> ##Here are additional operations: if n= a* m +b, then a=n // m and b = n%m, that is "a" is the floor division and "b" the remainder (also n modulo m)
>>> 50 // 3
16
>>> 50 % 3
2
>>>
```

Exercises:
50/3/2
30%4%3
40 x (34.5 -3)^2/4+3

Answers:
8.333333333333334
2
9925.5

- Assigning values of variables

- variable_name = value

```
>>> a = 5
>>> b = 3.2
>>> a+b
8.2
>>> a*b
16.0
>>> a//b
1.0
>>> a%b
1.7999999999999998
>>>
```

Cool feature: multiple simultaneous assignments

```
>>> x,y =2,3
>>> x1,y1,z2=1.0,3.4, 2+6.0j
>>> x
2
>>> y
3
# SWAP:
>>> x,y=y,x
>>> x
3
>>> y
2
######################
>>> x1,y1,z1=y1,z1,x1
>>> x1
3.4
>>> y1
(4+5j)
>>> z1
1.0
>>>
```

- Error if var. is not defined.

```
>>> a = 5
>>> b = 3.2
>>> a+b
8.2
>>> a*b
16.0
>>> a//b
1.0
>>> a%b
1.7999999999999998
>>> 4
4
>>> c
Traceback (most recent call last):
  File "<pyshell#39>", line 1, in <module>
    c
NameError: name 'c' is not defined
>>>
```

a^b??

```
>>> 10^9
3
>>> bin(10)
'0b1010'
>>> bin(9)
'0b1001'
>>> # a^b is BITWISE XOR: result is 1 if either the bit of a and of is 1, but not both. a+b moduo 2.
a+b%2
>>> bin(3)
'0b11'
>>>
```

Arithmetic Operations

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 30
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -10
* Multiplication	Multiplies values on either side of the operator	a * b = 200
/ Division	Divides left hand operand by right hand operand	b / a = 2
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 0
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands	9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0

Python comparison operators

For Example: a=10, b=20

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a!=b) is true
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

```
>>> a=10
>>> b=20
>>> a==b
False
>>> a!=b
True
>>> a>b
False
>>> a<b
True
>>> a>=b
False
>>> a<=b
True
>>>
```

Round-off errors

```
>>> .25+.25+.25 == .75
True
>>>
```

```
>>> 0.1+0.1==0.2
True
```

```
>>> 0.1 + 0.1 + 0.1 ==0.3
False
>>>
```

```
>>> 2**.5 * 2**.5 ==2
False
```

```
>>> 1/11
0.09090909090909091
>>> round(1/11,5)
0.09091
>>> round(1/11,6)
0.090909
>>>
```

```
>>> round(.1+.1+.1,10)==.3
True
>>> round(.1+.1+.1,20)==.3
False
```

```
>>> round((2**.5)* (2**.5) ,10)==2
True
>>> round((2**.5)* (2**.5) ,20)==2
False
```

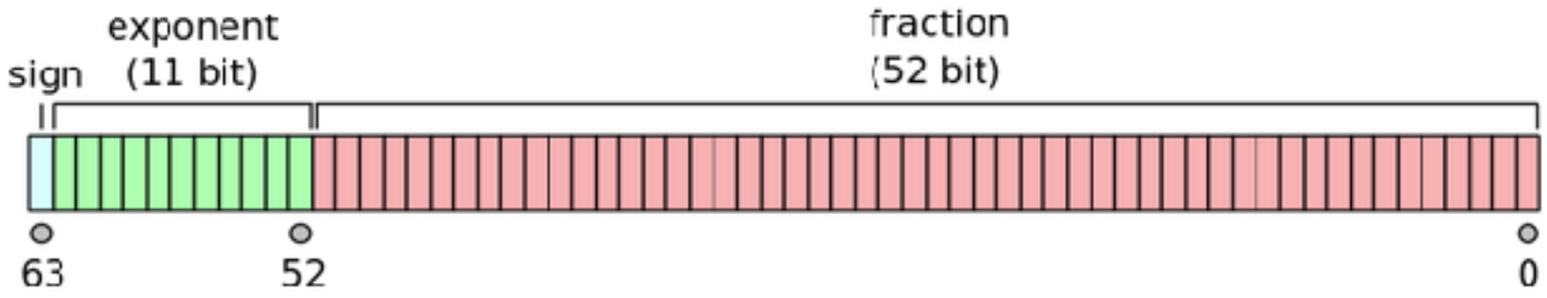
https://docs.python.org/3/tutorial/floatingpoint.html

Representation of reals (floats)

- Sign bit: 1 bit
- Exponent: 11 bits
- Significand precision: 53 bits (52 explicitly stored)

This gives from 15 to 17 significant decimal digits precision. If a decimal string with at most 15 significant digits is converted to IEEE 754 double-precision representation, and then converted back to a decimal string with the same number of digits, the final result should match the original string. If an IEEE 754 double-precision number is converted to a decimal string with at least 17 significant digits, and then converted back to double-precision representation, the final result must match the original number.[1]

The format is written with the significand having an implicit integer bit of value 1 (except for special data, see the exponent encoding below). With the 52 bits of the fraction significand appearing in the memory format, the total precision is therefore 53 bits (approximately 16 decimal digits, 53 $\log_{10}(2) \approx 15.955$). The bits are laid out as follows:



The real value assumed by a given 64-bit double-precision datum with a given biased exponent e and a 52-bit fraction is

$$(-1)^{\text{sign}}(1.b_{51}b_{50}\ldots b_0)_2 \times 2^{e-1023}$$

or

$$(-1)^{\text{sign}}\left(1+\sum_{i=1}^{52} b_{52-i}2^{-i}\right) \times 2^{e-1023}$$

https://en.wikipedia.org/wiki/Double-precision_floating-point_format

Or why 0.1 is not 1/10

Find J and N such that $J/2^N$ is closest to $1/10$ and J has 53 bits in its decimal expansion.

$$2^{52} < J < 2^{53} \implies \frac{1}{16} < \frac{J}{2^{56}} < \frac{1}{8} \implies N = 56.$$

$$2^{56} = 10J_1 + R_1$$

```
>>> 2**56//10
7205759403792793
>>> 2**56%10
6
>>>
```

$$\text{As } R_1 = 6, 2^{56} = 10(J_1 + 1) - 4, \implies J = J_1 + 1$$

$$\text{Closest number to 0.1 in Python} = +1 \times 7205759403792794 \times 2^{-56}$$

$$= \frac{3602879701896397}{36028797018963968}$$

```
>>> 2**(-100)
7.888609052210118e-31
>>> 2**(-1000)
9.332636185032189e-302
>>> 2**(-2000)
0.0
>>> 2**100
1267650600228229401496703205376
>>> 2.0**100
1.267650600228229e+30
>>> 2.0**1000
1.0715086071862673e+301
>>> 2.0**2000
```

Exercise: Find the smallest nonzero positive number and figure out why it is what it is. If 63 of the 64 bits were simply used to store the binary representation of the number, what is the smallest nonzero positive number and the largest?

Arbitrary precision integers

```
2**2000
Out[19]:
114813069527425452423283320117768198402231770208869520047764273682576626139237031385665948631650626991844596463898746277344711896086305533142593135616665318539129989145312280000688779148240044871428926990063486244781615463646388363947317026040466353970904996558162398808944629605623311649536164221970332681344168908984458505602379484807914058900934776500429002716706625830522008132236281291761267883317206598995396418127021779858404042159853183251540889433902091920554957783589672039160081957216630582755380425583726015528348786419432054508915275783882625175435528800822842770817965453762184851149029376
```

Classification of operators in Python

Arithmetic operators
Comparison (Relational) operators
Logical (Boolean) operators
Bitwise operators
Assignment operators
Special operators

```
>>> (2>1 and 3>2)
True
>>> (2>1 or 3>2)
True
>>> (2>1 or 3<2)
True
>>> (2>1 and 3<2)
False
>>> not (2>1)
False
>>>
```

Further assignment operators

=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5
**=	x **= 5	x = x ** 5

```
>>> x=3
>>> x += 2
>>> x
5
>>> x -=4
>>> x
1
>>> x *=5
>>> x
5
>>> x /=6
>>> x
0.8333333333333334
>>> x **=2
>>> x
0.6944444444444445
>>>
```

```
>>> x=2
>>> y=2
>>> id(x)
4297623968
>>> id(y)
4297623968
>>> x1=[1,2,3]
>>> y1=[1,2,3]
>>> id(x1)
4341064840
>>> id(y1)
4341066120
>>> x2=2.0
>>> y2=2.0
>>> id(x2)
4301331408
>>> id(y2)
4301331336
>>> x3='hello'
>>> y3='hello'
>>> id(x3)
4340992240
>>> id(y3)
4340992240
>>>
```

Strings and Lists

```
>>> a=3
>>> b=2.4
>>> c=4.0+3.0j
>>> s='Hello World'
>>> L=[a,b,c,s,7]
>>> # "a" is an integer, "b" a float, "c" a complex number, whlile "s" is a STRING ('string' or "string") and L is a LIST.
>>> type(a)
<class 'int'>
>>> type(b)
<class 'float'>
>>> type(c)
<class 'complex'>
>>> type(s)
<class 'str'>
>>> type(L)
<class 'list'>
>>> ##List is a collection that could be NONhomogeneous.
```

Elements of a list

```
>> L[0]
3
>>> L[1]
2.4
>>> L[2]
(4+3j)
>>> L[3]
'Hello World'
>>> L[4]
7
>>> L[5]
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    L[5]
IndexError: list index out of range
>>> L[0:2]
[3, 2.4]
>>> L[2:3]
[(4+3j)]
>>> L[2:4]
[(4+3j), 'Hello World']
>>>
```

L=[a,b,c,s,7]
L is one object whose 0-th element
or part is a=3,
2-nd element is a
complex number and so on.

```
>>> L=[a,b,s]
>>> L+L
[3, 2.4, 'Hello World', 3, 2.4, 'Hello World']
>>> 2*L
[3, 2.4, 'Hello World', 3, 2.4, 'Hello World']
>>> L+L==2*L
True
>>> L*L
Traceback (most recent call last):
  File "<pyshell#68>", line 1, in <module>
    L*L
TypeError: can't multiply sequence by non-int of type 'list'
>>> 2.0*L
Traceback (most recent call last):
  File "<pyshell#69>", line 1, in <module>
    2.0*L
TypeError: can't multiply sequence by non-int of type 'float'
>>> L2=[1,'Another list']
>>> L+L2
[3, 2.4, 'Hello World', 1, 'Another list']
>>> L2+L
[1, 'Another list', 3, 2.4, 'Hello World']
>>> L+L2==L2+L
False
>>>
```

Lists: Membership,Iteration …

```
>>> L=[1,2,4,8,9]
>>> 1 in L
True
>>> 3 in L
False
```

```
>>> for i in L:
        print(i)


1
2
4
8
9
>>> len(L)
5
```

```
>>> L=[1,2,4,8,9]
>>> L.append(3)
>>> L
[1, 2, 4, 8, 9, 3]
>>> l=[3,5,6]
>>> l+L
[3, 5, 6, 1, 2, 4, 8, 9, 3]
>>>
```

Strings attached

```
>>> s
'Hello World'
>>> s[0]
'H'
>>> s[5]
' '
>>> s[6:9]
'Wor'
>>> s+s
'Hello WorldHello World'
>>> s+' '+s
'Hello World Hello World'
>>>
```

Towards Programming

```
# Fibonacci Series using while:
# sum of two consecutive integers = next.
a,b=0,1
while b<10:
    print(b)
    a,b=b,a+b
```

- Use IDLE (or other means) to create fibonacc1.py with contents as above. “Run” it to get

```
========= RESTART: /Users/arul/Desktop/Python/Smallcodes/fibonacci1.py ========
1
1
2
3
5
8
>>>
```