Lecture2: Python Basics (Part I)

Data Science, DST, UIC

In this lecture, we will cover the following topics:

1. Python Syntax
2. Basic Data Types
3. Variables
4. Programming Styles
5. Operations '=-*/..."

1. Python Syntax

1.1 Python Indentation

Indentation refers to the **spaces at the beginning of a code line**.

Where in other programming languages the indentation in code is for readability only, the indentation in Python is **very important**.

- Python uses indentation to indicate **a block of code**.

In [1]:
```
score = 61
if score >= 60:
    print("Pass!")
else:
    print("Fail!")
```

```
Pass!
```

Python will give you an error if you skip the indentation:

In [2]:
```
score = 61
if score >= 60:
print("Pass!")  # no identation
else:
    print("Fail!")
```

```
  File "<ipython-input-2-459f1f4cad38>", line 3
    print("Pass!")  # no identation
        ^
IndentationError: expected an indented block
```

Python uses 4 spaces as indentation by default. However, the number of spaces is up to you, but a minimum of 1 space has to be used.

- You have to use the same number of spaces in the same block of code

In [3]:
```
score = 61
if score >= 60:
 print("Pass!") # use 1 space
else:
    print("Fail!") # use 4 spaces
```

```
Pass!
```

In [4]:
```
score = 61
if score >= 60:
 print("Pass!") # use 1 space
    print(":)") # use 4 spaces
else:
    print("Fail!")
```

```
  File "<ipython-input-4-66b47cc1ea84>", line 4
    print(":)") # use 4 spaces
    ^
IndentationError: unexpected indent
```

1.2 Python Variables

In python, variables are created when you assign a value to it. Python has no command for declaring a variable.

- Different from **Java** or **C** , you don't need to specify the type of a variable when declaring the variable.
- You can simply use assignment statement to declare a variable.

In [5]:
```
x = 5
y = "Hello World!"
x,y
```

Out[5]: (5, 'Hello World!')

1.3 Comments

Python has commenting capability for the purpose of in-code documentation. Comments start with a `#` , and Python will render the rest of the line as a comment:

In [6]:
```
# this is comment
print("Hello World!") # this is also a comment
```

```
Hello World!
```

Python does not really have a syntax for multi-line comments. To add a multi-line comment you could insert a `#` for each line:

In [7]:
```
# This is a comment
# written in
# more than just one line
print("Hello World!")
```

```
Hello World!
```

Or, not quite as intended, you can use a multi-line string.

- Since Python will ignore string literals that are not assigned to a variable, you can add a multiline string (triple quotes) in your code, and place your comment inside it:

In [8]:
```
"""
This is a comment
written in
more than just one line
"""
print("Hello World!")
```

```
Hello World!
```

2. Basic Data Types

More frequently used data types in python are the following:

- Numbers: int, float, complex number
- bool
- str
- list
- tuple
- set
- dict

2.1 Define and check the type of data

Variables can store data of different types, and different types can do different things. You can get the data type of any object by using the `type()` function

In [9]:
```
a = 1
type(1),type(a)
```

Out[9]: (int, int)

2.2 Numeric Types

There are three numeric types in Python:

- `int`
- `float`
- `complex`

Variables of numeric types are created when you assign a value to them:

In [10]:
```
a = 1
b = 0.1
c = 1j
```

Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

In [11]:
```
x = 1
y = -3255522
z = 11111
```

Float, or "floating point number" is a number, positive or negative, containing one or more decimals.

- Float can also be scientific numbers with an "e" to indicate the power of 10.

In [12]:
```
b = .6  #what does .6 mean?
print(b)
c = float(-7)
type(b),type(c)
```

```
0.6
```

Out[12]: (float, float)

In [13]:
```
p = 35e3
q = -12E4
p,q
```

Out[13]: (35000.0, -120000.0)

Complex numbers are numbers that look like `a+bj`, where `j` is the imaginary part

In [14]:
```
c = 3+5j
c, type(c)
```

Out[14]: ((3+5j), complex)

In [15]:
```
d = complex(5+3j) # create complex number using complex() function
c+d
```

Out[15]: (8+8j)

You can convert from one numeric type to another with the `int()`, `float()` and `complex()` functions.

In [17]:
```
x,y,z = 1, 1.8, 2j
a = float(x) # convert from int to float
b = int(y)   # convert from float to int
c = complex(x) # convert from int to complex
a,b,c
```

Out[17]: (1.0, 1, (1+0j))

2.3 bool

In Python, the two Boolean values are `True` and `False` (the capitalization must be exactly as shown), and the Python type is `bool`.

In [20]:
```
type(True) #is "true" ok?
# type(true)
type(False)
```

Out[20]: bool

2.4 Strings

Strings are defined either with a **single quote** or a **double quotes** and the python type is `str`.

In [21]:
```
type('data science')
```

Out[21]: str

In [22]:
```
type("Data Processing Workshop")
```

Out[22]:
```
str
```

You can assign a multiline string to a variable by using three quotes:

In [23]:
```
a = """This is a string with
multiple lines."""
a, type(a)
```

Out[23]:
```
('This is a string with \nmultiple lines.', str)
```

Strings in Python are arrays of characters. However, Python does not have a character data type, a single character is simply a string with length 1. Square brackets can be used to access elements of the string:

In [24]:
```
a = "Hello World!"
a[0]
```

Out[24]:
```
'H'
```

You can use the `+` operator to concatenate two strings:

In [25]:
```
a = "Hello"
b = "World"
c = a+" "+b + "!"
c
```

Out[25]:
```
'Hello World!'
```

2.5 Lists

Lists are very similar to arrays, which are used to store multiple items in a single variable.

- They can contain any type of variable, and they can contain as many variables as you wish.
- use **[ ]** to express a list in python

Lists are one of 4 built-in data types in Python used to store collections of data, the other three are `Tuple`, `Set` and `Dictionary`, all with different qualities and usage.

In [26]:
```
al = [1,2,3,4,5]
print(type(al))
type([1,0.2, True, "Hello"])
```

```
<class 'list'>
```

Out[26]:
```
list
```

List items are ordered, changeable, and allow duplicate values.

- If you add new items to a list, the new items will be placed at the end of the list.
- The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.
- Lists can have items with the same value.

List items are indexed, the first item has index [0], the second item has index [1] etc.

In [27]:
```
testList = ["a","b","a","c"]
testList, len(testList) # use len() function to determin how may items a lsit has
```

Out[27]:
```
(['a', 'b', 'a', 'c'], 4)
```

In [28]:
```
print(testList[1]) # second item
print(testList[-2])# second last item
testList[2] = "d" # change item value
testList
```

```
b
a
```

Out[28]:
```
['a', 'b', 'd', 'c']
```

2.6 Tuple

A tuple is a sequence of **immutable** Python objects. Tuples are sequences, just like lists.

- use **( )** to express a tuple in python
- Tuple items are indexed, the first item has index [0], the second item has index [1] etc.
- Tuples are unchangeable, meaning that we cannot change, add or remove, items after the tuple has been created.
- Tuples allow duplicates.

In [29]:
```
at = (1,2,3,4,5)
print(type(at))
type((1,.2,True))
```

```
<class 'tuple'>
```

Out[29]:
```
tuple
```

In [30]:
```
testTuple = (1,2,"a","a",3,4)
testTuple[0],len(testTuple)
```

Out[30]:
```
(1, 6)
```

To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple.

In [31]:
```
oneItemTuple = ("one",)
oneItem = ("one")

type(oneItemTuple),type(oneItem)
```

Out[31]:
```
(tuple, str)
```

2.7 Set

A set is a collection of items which is unordered and unindexed.

- use **{ }** to express a set in python

Sets cannot have two items with the same value. Therefore, duplicate values will be **ignored**.

In [32]:
```
ase = {1,2,3,4,5}
a = {1,1,2,3} # duplicate values will be ignored
type(ase), ase, a
```

Out[32]:
```
(set, {1, 2, 3, 4, 5}, {1, 2, 3})
```

To determine how many items a set has, use the `len()` method.

In [33]:
```
print(len(a))
```

```
3
```

2.8 Dictionary

A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they store data values in `key:value` pairs.

- use **{:,}** to express a dict, which is a special type of set with **Keys**

In [34]:
```
ad = {"a":0, "b":1, "c":2}
type(ad)
```

Out[34]:
```
dict
```

Dictionary items are presented in `key:value` pairs, and can be referred to by using the key name.

In [35]:
```
ad["a"]
```

Out[35]:
```
0
```

Dictionaries cannot have two items with the same key:

- Duplicate values will overwrite existing values:

In [36]:
```
testDict = {"a":0, "b":1,"a":"c"}
testDict
```

Out[36]:
```
{'a': 'c', 'b': 1}
```

- The `keys()` method will return a list of all the keys in the dictionary.
- The `values()` method will return a list of all the values in the dictionary.
- The `items()` method will return each item in a dictionary, as tuples in a list.

In [37]:
```
testDict.keys(), testDict.values(), testDict.items()
```

Out[37]:
```
(dict_keys(['a', 'b']),
 dict_values(['c', 1]),
 dict_items([('a', 'c'), ('b', 1)]))
```

You can change the value of a specific item by referring to its key name

- If the key exists, the value will be changed
- If the key does not exist, a new key:value pair will be added to the dictionary

In [38]:
```
testDict['a'] = 10
print(testDict)
testDict['new'] = "new"
testDict
```

```
{'a': 10, 'b': 1}
```

Out[38]:
```
{'a': 10, 'b': 1, 'new': 'new'}
```

Python is an **object-oriented programming** language, and in Python everything is an object.The data types we have discussed so far, including int,float,...,set,dictionary, are object.

2.9 Data type conversion

To judge whether a data or variable is a particular type, use the function `isinstance()`

In [39]:
```
x=100
print(isinstance(x, int))

y=100.0
print(isinstance(y, int))

# In python, bool type is a subclass of int
print(isinstance(True, int))

print(isinstance('a',int))
```

```
True
False
True
False
```

Python defines type conversion functions to directly convert one data type to another. Python has two types of type conversion:

- Implicit type conversion (among int, float, complex, bool)
- Explicit type conversion

In Implicit type conversion, Python automatically converts one data type to another data type. This process doesn't need any user involvement.

In [40]:
```
num_int = 123
num_flo = 1.23
num_new = num_int + num_flo + True
print(num_new)
type(num_new)
```

```
125.23
```

Out[40]:
```
float
```

we can see the `num_new` has float data type because Python always converts smaller data type to larger data type to avoid the loss of data.

In [41]:
```
num_str = "456"
num_ns = num_int + num_str
print(num_ns)
type(num_ns)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-41-72b1366e973c> in <module>()
      1 num_str = "456"
----> 2 num_ns = num_int + num_str
      3 print(num_ns)
      4 type(num_ns)

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

Python is not able use implicit conversion in such condition. Python has the solution for this type of situation which is know as explicit conversion.

In explicit type conversion, users convert the data type of an object to required data type. We use the predefined functions like `int()`, `float()`, `str()`, etc to perform explicit type conversion.

- Syntax: `required_datatype(expression)`

In [42]:
```
print(int(2.5))
print(float(100))
```

```
2
100.0
```

Number 0 can be converted to bool type `False`, and other non-zero values can be converted to `True`.

In [43]:
```
int(True),int(False)
```

Out[43]: (1, 0)

In [44]:
```
bool(0.0),bool('abc')
```

Out[44]: (False, True)

You can convert between list and tuple. List and tuple look similar, however they are different.

- Lists are **mutable** and tuples are **immutable**.

In [45]:
```
tuple([3,1,4])
```

Out[45]: (3, 1, 4)

In [46]:
```
list((1,4,7))
```

Out[46]: [1, 4, 7]

Conversion between number systems

- `int(a,base)` : This function converts any data type to integer. **Base** specifies the base in which string is if data type is string.

In [47]:
```
print(int("1000"))
int('1000',base=2)
```

```
1000
```

Out[47]: 8

In [48]:
```
int('FF',base=16)
```

Out[48]: 255

In [49]:
```
int('99',base=10)
```

Out[49]: 99

3. Variables

- Variables are containers for storing data values.
- Naming of variable should follow the rules:
  1. Variable name includes only **alphabets, nubmers and underscore '_'**
  2. Variable name cannot start with **numbers**
  3. Please do not use **keywords** in python as the variable names, unless you really know what you are doing
  4. Variable names are case-sensitive

In [50]:
```
# legal variable names
_test0 = 1
test_0 = 2
_test_0 = 3
test0 = 4
Test0 = 5
testVar = 6
```

In [51]:
```
i = 'i'
I
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-51-a188e4705666> in <module>()
      1 i = 'i'
----> 2 I

NameError: name 'I' is not defined
```

In [52]:
```
# illegal variable names
0_test = 7  # start with number
test-0 = 8  # contains "-"
```

```
  File "<ipython-input-52-3585920c67f3>", line 2
    0_test = 7  # start with number
    ^
SyntaxError: invalid token
```

In [53]:
```
True = 2 # True is a keyword
```

```
  File "<ipython-input-53-16318898c997>", line 1
    True = 2 # True is a keyword
         ^
SyntaxError: can't assign to keyword
```

Method to check all the **keywords** in python

In [54]:
```
import keyword
keyword.kwlist
```

Out[54]:
```
['False',
 'None',
 'True',
 'and',
 'as',
 'assert',
 'async',
 'await',
 'break',
 'class',
 'continue',
 'def',
 'del',
 'elif',
 'else',
 'except',
 'finally',
 'for',
 'from',
 'global',
 'if',
 'import',
 'in',
 'is',
 'lambda',
 'nonlocal',
 'not',
 'or',
 'pass',
 'raise',
 'return',
 'try',
 'while',
 'with',
 'yield']
```

Python allows you to assign values to multiple variables in one line:

In [55]:
```
testBool, testInt, testFloat, testStr = True, 20, 10.6, "MyStr"
testBool,testInt,testFloat,testStr
```

Out[55]:
```
(True, 20, 10.6, 'MyStr')
```

Python is **dynamic** typed language, meaning you can change the type of a variable by directly assignment a value of different type to it.

In [56]:
```
x=100
print(type(x))
x="bite me"
print(type(x))
x
```

```
<class 'int'>
<class 'str'>
```

Out[56]:
```
'bite me'
```

Python is **Strong** typed language, meaning type conversion should be made **explicitly**, except among **float**, **bool**, **int**, **complex**

```
In [57]: '3'+2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-57-aacce49a0307> in <module>()
----> 1 '3'+2

TypeError: can only concatenate str (not "int") to str
```

```
In [58]: int('3')+2
```

```
Out[58]: 5
```

Method to check all the variable names used so far

In [59]:
```
dir()
```

Out[59]: ['In',
'Out',
'Test0',
'',
'_',
'_12',
'_13',
'_14',
'_15',
'_16',
'_17',
'_18',
'_20',
'_21',
'_22',
'_23',
'_24',
'_25',
'_26',
'_27',
'_28',
'_29',
'_30',
'_31',
'_32',
'_34',
'_35',
'_36',
'_37',
'_38',
'_40',
'_43',
'_44',
'_45',
'_46',
'_47',
'_48',
'_49',
'_5',
'_54',
'_55',
'_56',
'_58',
'_9',
'__',
'___',
'__builtin__',
'__builtins__',
'__doc__',
'__loader__',
'__name__',
'__package__',
'__spec__',
'_dh',
'_i',
'_i1',
'_i10',
'_i11',
'_i12',
'_i13',
'_i14',
'_i15',
'_i16',
'_i17',
'_i18',
'_i19',
'_i2',
'_i20',
'_i21',

'_i22',
'_i23',
'_i24',
'_i25',
'_i26',
'_i27',
'_i28',
'_i29',
'_i3',
'_i30',
'_i31',
'_i32',
'_i33',
'_i34',
'_i35',
'_i36',
'_i37',
'_i38',
'_i39',
'_i4',
'_i40',
'_i41',
'_i42',
'_i43',
'_i44',
'_i45',
'_i46',
'_i47',
'_i48',
'_i49',
'_i5',
'_i50',
'_i51',
'_i52',
'_i53',
'_i54',
'_i55',
'_i56',
'_i57',
'_i58',
'_i59',
'_i6',
'_i7',
'_i8',
'_i9',
'_ih',
'_ii',
'_iii',
'_oh',
'_test0',
'_test_0',
'a',
'ad',
'al',
'ase',
'at',
'b',
'c',
'd',
'exit',
'get_ipython',
'i',
'keyword',
'num_flo',
'num_int',
'num_new',
'num_str',
'oneItem',

```
'oneItemTuple',
'p',
'q',
'quit',
'score',
'test0',
'testBool',
'testDict',
'testFloat',
'testInt',
'testList',
'testStr',
'testTuple',
'testVar',
'test_0',
'x',
'y',
'z']
```

4. Programming conventions

Please refer to

PEP8-Style Guide for Python Code (https://legacy.python.org/dev/peps/pep-0008/)

- One statement in one line
- NO "\;" sign in the end, not like C or Java

In [60]:
```
first = 1
second = first+1
third = 3
```

If you have to write multiple statements in one line, use ";" to seperate them

In [61]:
```
i=1;j=2;k=3
i;j;k   # different from i,j,k, because i,j,k actually means a tuple
```

Out[61]: 3

In [62]:
```
i,j,k
```

Out[62]: (1, 2, 3)

Sometimes it looks better if you seperate a statement into multiple lines.

In this case, use "\" to indicate the line seperation

In [63]:
```
print("hello data \
science")
1+\
2
```

```
hello data science
```

Out[63]: 3

5. Operators

Python Arithmetic Operators

Arithmetic operators are used with numeric values to perform common mathematical operations:

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

In [64]:
```
# Examples of Arithmetic Operators

a = 9
b = 4

# Addition of numbers
add = a + b
# Subtraction of numbers
sub = a - b
# Multiplication of number
mul = a * b
# Division(float) of number
div1 = a / b
# Division(floor) of number
div2 = a // b
# Modulo of both number
mod = a % b

# print results
print(add)
print(sub)
print(mul)
print(div1)
print(div2)
print(mod)
```

```
13
5
36
2.25
2
1
```

Python Assignment Operators

Assignment operators are used to assign values to variables:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Python Comparison Operators

Comparison operators are used to compare two values:

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

In [65]:
```
# Examples of Comparison Operators


a = 13
b = 33

# a > b is False
print(a > b)

# a < b is True
print(a < b)

# a == b is False
print(a == b)

# a != b is True
print(a != b)

# a >= b is False
print(a >= b)

# a <= b is True
print(a <= b)
```

```
False
True
False
True
False
True
```

Python Logical Operators

Logical operators are used to combine conditional statements:

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

In [66]:
```
# Examples of Logical Operators

a = True
b = False

# Print a and b is False
print(a and b)

# Print a or b is True
print(a or b)

# Print not a is False
print(not a)
```

```
False
True
False
```

Python Identity Operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location

Operator	Description	Example
is	Returns true if both variables are the same object	x is y
is not	Returns true if both variables are not the same object	x is not y

In [67]:
```
# Examples of Identity operators

a1 = 3
b1 = 3
a2 = 'DataScience'
b2 = 'DataScience'

a3 = 'Data Science'
b3 = 'Data Science'

a4 = [1,2,3]
b4 = [1,2,3]

# small numbers in python share the same object ([-3,256])
print(a1 is b1)

# strings without special character share the same object;
# strings with special chararacter doesn't share same object
print(a2 is b2)
print(a3 is b3)

print(a4 is b4)
```

```
True
True
False
False
```

Python Membership Operators

Membership operators are used to test if a sequence is presented in an object:

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

In [68]:
```
# Examples of Membership operators

x = 'Data Science'
y = {3:'a',4:'b'}

print('D' in x)

print('workshop' not in x)

print('Sci' not in x)

print(3 in y)

print('b' in y)
```

```
True
True
False
True
False
```

Python Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

In [65]:
```
# Examples of Bitwise operators

a = 10  # 0...1010  (all zeros left)
b = 4   # 0...0100  (all zeros left)

# Print bitwise AND operation
print(a & b)   # 0...0000

# Print bitwise OR operation
print(a | b) # 000....1110  - 14

# Print bitwise NOT operation
print(~a)  # 11111....0101


# print bitwise XOR operation
print(a ^ b) # 000...1110

# print bitwise right shift operation
print(a >> 2) # 0000...10

# print bitwise left shift operation
print(a << 2) # 00000...101000
```

```
0
14
-11
14
2
40
```

Precednce of operators

Precedence	Associativity	Operator	Description
18	Left-to-right	()	Parentheses (grouping)
17	Left-to-right	*f*(args...)	Function call
16	Left-to-right	*x*[index:index]	Slicing
15	Left-to-right	*x*[index]	Array Subscription
14	Right-to-left	**	Exponentiation
13	Left-to-right	~*x*	Bitwise not
12	Left-to-right	+*x* -*x*	Positive, Negative
11	Left-to-right	* / %	Multiplication Division Modulo
10	Left-to-right	+ -	Addition Subtraction
9	Left-to-right	<< >>	Bitwise left shift Bitwise right shift
8	Left-to-right	&	Bitwise AND
7	Left-to-right	^	Bitwise XOR
6	Left-to-right	|	Bitwise OR
5	Left-to-right	in, not in, is, is not, <, <=, >, >=, <>, == !=	Membership Relational Equality Inequality
4	Left-to-right	not *x*	Boolean NOT
3	Left-to-right	and	Boolean AND
2	Left-to-right	or	Boolean OR
1	Left-to-right	lambda	Lambda expression

Try it out for yourself!